**UNIVERSIDAD AUTONOMA DE MADRID**

**ESCUELA POLITECNICA SUPERIOR**



**Grado en Ingeniería Informática**

# TRABAJO FIN DE GRADO

## Arquitecturas serverless: qué son y a dónde nos llevan

**Javier Encinas Cortés**
**Tutor: Roberto Latorre Camino**

**junio 2020**

# ARQUITECTURAS SERVERLESS:
# qué son y a dónde nos llevan

**AUTOR: Javier Encinas Cortés**
**TUTOR: Roberto Latorre Camino**

**Escuela Politécnica Superior**
**Universidad Autónoma de Madrid**
**junio de 2020**

# *Agradecimientos*

En primer lugar, se lo quiero agradecer a Roberto Latorre Camino por su implicación, su dedicación y su tiempo, ya que sin él esa pequeña idea que me rondaba la cabeza no habría cobrado forma. Aprovecho también para agradecer a todos los docentes que han dado todo de sí, su labor y dedicación para que podamos aprender y ser mejores personas, haciendo especial mención a Idoia Alarcón que siempre me ha ofrecido su ayuda y ha sido un gran apoyo durante el grado.

Se lo agradezco a mi familia, especialmente a mi hermano y a mis abuelos, que me han apoyado en todo momento y me han dado los medios para ser lo que soy hoy.

Se lo agradezco a mis compañeros y amigos tanto del grado como de la asociación, con los que sin su apoyo no hubiese sido posible terminar el grado, y, a día de hoy son un pilar fundamental en mi vida.

Por último, hacer una mención especial a mi pareja, Ana Elsa, que ha estado a mi lado todos los años del grado, que me ha apoyado en las mejores y peores situaciones y que sin ella no estaría donde estoy.

Este Trabajo de Fin de Grado va dedicado especialmente a mis abuelos, a mi pareja y amigos.

# Resumen

Las **arquitecturas *serverless*** (literalmente, “sin servidor”) son un nuevo paradigma para construir y operar con sistemas informáticos que nacen en 2010 y cobran fuerza tras la salida de *AWS Lambda* a manos de *Amazon Web Services* (*AWS*). Lo novedoso de este modelo arquitectónico es que se caracteriza por eliminar la administración del lado del servidor, delegando esta responsabilidad en el proveedor de servicio o en un *framework cloud*, de forma que el desarrollador se puede centrar en el desarrollo del producto sin preocuparse de la administración, lo que reduce el *time to market*, es decir, el tiempo desde que se concibe la idea hasta que se está produciendo. La industria está adoptando rápidamente las arquitecturas *serverless,* ya que tienen un alto nivel de acoplamiento que favorece el desarrollo de arquitecturas basadas en microservicios, cobrando así una gran importancia en la nube pública y privada, y haciendo que estos modelos sean una tecnología puntera que no se ven en el grado y, a pesar de ser tan novedosa hay pocas fuentes realmente fiables.

Este Trabajo Fin de Grado explica qué son las arquitecturas *serverless* y sus principales características, ventajas y desventajas frente a otros modelos, de forma que pueda ser útil para cualquiera que quiera formarse y entender cómo funcionan estas arquitecturas o quiera desarrollar un proyecto con las mismas. Para ello, se han revisado, analizado y contrastado las descripciones y referencias más relevantes de la literatura, identificando diversos conceptos o aspectos que habitualmente se utilizan de manera ambigua en la propia bibliografía de referencia. Además, se aportan explicaciones y evidencias adicionales obtenidas durante el desarrollo del trabajo al realizar diversas pruebas de concepto con arquitecturas *serverless* desplegadas en AWS. Algunas de estas evidencias apoyan las afirmaciones que se pueden leer en la documentación sobre las arquitecturas *serverless*, pero otras parecen indicar que alguna de las afirmaciones que habitualmente se toman como certeza, realmente no lo son. Por lo que en todo momento, las explicaciones incluyen aportaciones que se han formado durante el trabajo, estudiando estas arquitecturas, realizando pruebas y contrastando datos.

Para poder explicar todo lo anterior en detalle, el trabajo comienza contextualizando de dónde surgen las primeras ideas referentes a almacenamiento de datos centralizado y tecnologías de tiempo compartido, las cuales son la esencia de las tecnologías en la nube que dan lugar a las arquitecturas *serverless* y finalmente cómo cobran fuerza estas arquitecturas a manos de grandes empresas. También se explican conceptos básicos de las arquitecturas *cloud* y los diferentes tipos de servicios *cloud* para poder explicar este modelo. Esto nos conduce al núcleo del trabajo, donde se explica qué es el modelo *serverless* y sus características principales con ayuda de ejemplos para su mejor comprensión. Después, se enumeran y discuten las ventajas y las desventajas que componen estas arquitecturas como un precedente a la conclusión del trabajo.

Un aspecto importante a destacar es que durante el desarrollo del trabajo se identificó que gran parte de las fuentes de información sobre *serverless* podían estar sesgadas, ya bien por el atractivo de estas arquitecturas, por estar influenciados por los proveedores de servicio o por ser documentación oficial del propio proveedor, además de poder contener errores o información imprecisa, por ello, se ha realizado una lectura crítica tratando de

contrastar todas las afirmaciones con fuentes oficiales, y en la medida de lo posible, realizar pruebas de concepto que prueben tales afirmaciones.


# Palabras clave (castellano)

Arquitectura de sistemas, Arquitecturas *Serverless*, Arquitecturas cloud, Computación en la nube, tecnologías de tiempo compartido.

# Abstract (English)

Serverless architectures are a new paradigm for building and operating with computer systems that were born in 2010 and gain strength after the release of *AWS Lambda* from *Amazon Web Services (AWS).* The novelty of this architectural model is that it is characterized by eliminating server-side administration, delegating this responsibility to the service provider or a cloud framework, this allows to the developer to focus on product development without worrying about the administration which reduces the time to market, that is, the time from the idea is conceived until it is being produced. The industry is rapidly adopting serverless architectures, since they have a high level of coupling that favors the development of architectures based on micro-services, thus gaining great importance in the public and private cloud, and making these models a cutting-edge technology that does not it is seen in the degree and, despite being so novel there are few really reliable sources.

This Bachelor Thesis explains what serverless architectures are and their main characteristics, advantages and disadvantages compared to other models, so that it can be useful for anyone who wants to train and understand how these architectures work or want to develop a project with them. For this, the most relevant descriptions and references in the literature have been reviewed, analyzed and contrasted, identifying various concepts or aspects that are usually used ambiguously in the reference bibliography itself. In addition, additional explanations and evidence obtained during the development of the Bachelor Thesis are provided by carrying out various proofs of concept with serverless architectures deployed in AWS. Some of this evidence supports the statements that can be read in the documentation about serverless architectures, but others seem to indicate that some of the statements that are usually taken as certainty, really are not. So, at all times, the explanations include contributions that have been formed during the Bachelor Thesis, studying these architectures, conducting tests and contrasting data.

To explain all of the above in detail, the Bachelor Thesis begins contextualizing where these first ideas regarding centralized data storage and timeshare technologies, which are the essence of cloud technologies that give rise to serverless and finally how these architectures gain strength at the hands of large companies. As well, it is explained basic concepts of cloud architectures and the services that theses provide to explain this model. These lead us to the core of the bachelor where it is explained what is serverless and its main characteristics with the help of some examples for a better understanding. Then the advantages and disadvantages that make up these architectures are listed and discussed as a precedent to the conclusion of the bachelor and that provide a perspective to everything related.

An important aspect related with this Bachelor is that during the development of this Bachelor Thesis it was identified that a large part of the sources of *serverless* could be biased, either because of the attractiveness of these architectures because they are influenced by service providers or because they are official documentation from the provider itself, in addition to being able to contain errors or imprecise information, therefore, a critical reading has been made trying to contrast all the claims with official sources, and to the extent possible, carry out proofs of concept to prove such claims.

## Keywords (inglés)

System architectures, Serverless architectures, Cloud architectures, Cloud computing, Timeshare technologies.

# ÍNDICE DE CONTENIDOS


1 Introducción ..... 1

1.1 Motivación ..... 1
1.2 Objetivos ..... 3
1.3 Organización de la memoria ..... 4
2 Estado del arte ..... 7

2.1 Contexto histórico ..... 7
2.2 Servicios en la nube ..... 8
2.2.1 Tipos de servicios en la nube ..... 8
2.2.2 Proveedores de servicio, nube pública, nube privada, nube híbrida y nube comunitaria ..... 10
3 Diseño ..... 13

4 Comprendiendo *serverless* ..... 14

4.1 ¿Qué es *serverless*? ..... 14
4.1.1 Comprendiendo *serverless*, ejemplo 1 ..... 14
4.1.2 Comprendiendo *serverless*, ejemplo 2 ..... 16
5 Desarrollo ..... 18

5.1 Propiedades de las arquitecturas *serverless* en detalle ..... 18
5.1.1 Estado de la aplicación ..... 19
5.1.2 Duración de la ejecución ..... 19
5.1.3 Latencia de arranque, arranques en frío y arranques en caliente 19
6 Discusión ..... 21

6.1 API Gateway ..... 21
6.2 Comparación entre PaaS y FaaS ..... 21
6.3 Comparación entre sistemas de microservicios con contenedores y arquitecturas *serverless* ..... 22
6.4 AWS Fargate ..... 23
6.5 Stored Procedure as a service y *FaaS* ..... 23
6.6 Ventajas ..... 24
6.6.1 Reducción de costes ..... 24
6.6.1.1 Reducción de costes operacional ..... 24
6.6.1.2 Reducción de costes de desarrollo ..... 25
6.6.1.3 Reducción de costes de escalado ..... 25
6.6.1.4 Optimización como medida de reducción de costes ..... 27
6.6.2 Facilidad de gestión operacional ..... 27
6.6.2.1 Beneficios operacionales de escalado ..... 27
6.6.2.2 Reducción de la complejidad de empaquetado y despliegue ..... 27
6.6.2.3 Experimentación continua y *time to market* ..... 28
6.6.3 Computación más ecológica ..... 28
6.7 Inconvenientes ..... 29
6.7.1 Inconvenientes inherentes al modelo ..... 29
6.7.1.1 Control del proveedor ..... 29

6.7.1.2 Inconvenientes de tecnologías de uso compartido ..... 29
6.7.1.3 Dificultad para migrar y dependencia del proveedor ..... 30
6.7.1.4 Aumento del riesgo en la seguridad informática ..... 30
6.7.1.5 Pérdida de optimización en el lado del servidor ..... 31
6.7.1.6 No se conserva el estado ..... 31
6.7.2 Inconvenientes de implementación ..... 31
6.7.2.1 Denegación de servicio ..... 32
6.7.2.2 Duración de la ejecución limitada ..... 32
6.7.2.3 Latencia de arranque ..... 32
*6.7.2.4* Testing en arquitecturas *serverless* ..... 33
*6.7.2.5* Carencias en la depuración de errores ..... 33
6.7.2.6 Monitorización ..... 34
6.7.2.7 Subestimar u olvidar la operativa ..... 34
7 Pruebas y resultados ..... 35
8 Conclusiones y trabajo futuro ..... 36
8.1 Conclusiones ..... 36
8.2 Trabajo futuro ..... 37
Referencias ..... I
Glosario ..... III
Anexos ..... I
A. Pruebas de concepto y resultados ..... I
Arranques en frío y en caliente ..... I
Duración de la ejecución ..... III
Estado ..... V
Escalado automático ..... VI

# ÍNDICE DE FIGURAS

FIGURA 1. WHERE WILL WORKLOADS RUN (TODAY VERSUS 2020) [12]. ..... 2
FIGURA 2. CAPAS DE LA COMPUTACIÓN EN LA NUBE. [17] ..... 9
FIGURA 3. GRADO DE AUTOMATIZACIÓN USANDO SERVERLESS. [18] ..... 10
FIGURA 4. MAGIC QUADRANT FOR CLOUD INFRASTRUCTURE AS A SERVICE WORLDWIDE. [19] ..... 11
FIGURA 5. ARQUITECTURA TRADICIONAL SIMPLE DE UNA APLICACIÓN DE TIPO TIENDA. ..... 15
FIGURA 6. ARQUITECTURA *SERVERLESS* DE UNA APLICACIÓN DE TIPO TIENDA. ..... 15

FIGURA 7. DIAGRAMA DE ARQUITECTURA TRADICIONAL DE UN SISTEMA DE SUGERENCIAS. .... 17

FIGURA 8.DIAGRAMA DE ARQUITECTURA *SERVERLESS* DE UN SISTEMA DE SUGERENCIAS. ..... 17

FIGURA 9. CÓDIGO DE LA FUNCIÓN A PROBAR LATENCIA DE ARRANQUE ..... II

FIGURA 10. GRÁFICA DE DURACIÓN MÁXIMA, MEDIA Y MÍNIMA DE LA EJECUCIÓN. ..... III

FIGURA 11. CÓDIGO DE PRUEBA PARA COMPROBAR LA DURACIÓN DE LA EJECUCIÓN DE UNA FUNCIÓN *LAMBDA*. ..... IV

FIGURA 12. CÓDIGO USADO PARA PROBAR SI SE TRATAN DE FUNCIONES SIN ESTADO. ..... V

FIGURA 13. GRÁFICA DE EJECUCIONES CONCURRENTES. ..... VIII

# ÍNDICE DE TABLAS

TABLA 1. INGRESOS MUNDIALES DE SERVICIOS DE NUBE PÚBLICA (MILLONES DE DÓLARES) [11]. ..... 1

TABLA 2. CARACTERÍSITICAS DE LAS TECNOLOGÍAS *FAAS* DE LOS PRINCIPALES PROVEEDORES DE NUBE PÚBLICA. [4] ..... 11

TABLA 3. COMPARACIÓN ENTRE LAS PRINCIPALES VENTAJAS Y DESVENTAJAS DE LAS ARQUITECTURAS *SERVERLESS.* ..... 37

TABLA 4. REGISTROS DE LAS 10 EJECUCIONES DE PRUEBA. ..... II

TABLA 5.TABLA RESUMEN DE LAS CARACTERÍSTICAS DE LAS PRUEBAS. ..... II

TABLA 6. REGISTRO DE LA EJECUCIÓN DE LA PRUEBA CON UN LÍMITE DE 3 SEGUNDOS. ..... IV

TABLA 7. REGISTRO DE LA EJECUCIÓN DE LA PRUEBA CON UN LÍMITE DE 15 MINUTOS. ..... IV

TABLA 8. TABLA RESUMEN DE LA EJECUCIÓN DE LA PRUEBA CON UN LÍMITE DE 15 MINUTOS. ... V

TABLA 9. REGISTRO DE LA PRIMERA EJECUCIÓN PROBANDO EL ESTADO DE *AWS LAMBDA*. .... VI

TABLA 10.REGISTRO DE LA SEGUNDA EJECUCIÓN PROBANDO EL ESTADO DE AWS LAMBDA. .. VI

TABLA 11. REGISTRO DE LA PRIMERA EJECUCIÓN CONCURRENTE DE LA PRUEBA DE ESCALADO HORIZONTAL. ..... VII

TABLA 12. REGISTRO DE LA SEGUNDA EJECUCIÓN CONCURRENTE DE LA PRUEBA DE ESCALADO HORIZONTAL. ..... VIII

# 1 Introducción

La computación en la nube o *cloud computing* supone un avance tecnológico que hace unos años no podíamos imaginar, a día de hoy está provocando una transformación muy importante que está cambiando la estrategia tecnológica de las empresas. De la computación en la nube cobra fuerza el modelo de arquitectura *serverless*, una tecnología que despierta gran interés al sector de la informática y de las empresas tecnológicas.

Este Trabajo de Fin de Grado (TFG) tiene como objetivo, en términos generales, describir y estudiar las características del modelo *serverless*.

## 1.1 Motivación

Esta memoria de TFG basa sus **principales motivaciones** en el crecimiento de las tecnologías en la nube o también llamadas tecnologías *cloud* y el interés por estas tecnologías y la computación en la nube. La llegada de las tecnologías en la nube ha supuesto un gran cambio en la informática, según un estudio de 2018 presentado en *BrightTALK* [9], 451 Research estimaba que el 90% de las empresas tecnológicas usan estas tecnologías. Sumado a lo anterior, un estudio realizado por la revista *Forbes* en 2018 aseguraba que aproximadamente el 77% de las empresas tienen al menos una aplicación o una porción de la empresa en la nube [10]. Esto claramente supone un cambio en la informática proporcionando alternativas y suponiendo una competencia a las arquitecturas más tradicionales, donde las empresas tenían que invertir grandes cantidades en centros de procesamiento de datos o CPD y servidores *Rack*, cambiando en gran medida el modelo de negocio del apartado tecnológico de las empresas donde arquitecturas como *serverless* entran en juego, aprovechando la fuerza de las tecnologías *cloud* y ofreciendo grandes ventajas.

La confianza que depositan las empresas en los proveedores de servicios en la nube y la gran cantidad de productos que éstos ofrecen, hacen del mercado de las tecnologías en la nube uno de los más interesantes y rentables hoy en día. En 2019 según *Gartner*, empresa consultora líder mundial en investigación y asesoramiento, este mercado ingresó 214.3 billones de dólares, y en 2020 se prevé que llegue a ingresar 249.8 billones de dólares [11] (ver evolución en la Tabla 1).

| | 2018 | 2019 | 2020 | 2021 | 2022 |
|---|---|---|---|---|---|
| Cloud Business Process Services (BPaaS) | 45.800 | 48.300 | 53.100 | 57.00 | 61.100 |
| Cloud Application Infrastructure Services (PaaS) | 15.600 | 19.000 | 23.000 | 27.500 | 31.800 |
| Cloud Application Services (SaaS) | 80.000 | 94.800 | 110.500 | 126.700 | 143.700 |
| Cloud Management and Security Services | 10.500 | 12.200 | 14.100 | 61.900 | 76.600 |
| Cloud System Infrastructure Services (IaaS) | 30.500 | 38.900 | 49.100 | 61.900 | 76.600 |
| Total | 182.400 | 214.300 | 249.800 | 289.100 | 331.200 |

Tabla 1. Ingresos mundiales de servicios de nube pública (millones de dólares) [11].

La revista *Forbes* publicaba un artículo en 2018 [12] en el que afirmaba que el 83% de las cargas de trabajo del sector empresarial se encontrarían en la nube en 2020 según una encuesta realizada por *LogicMonitor's*. Además, la revista también exponía según otra encuesta realizada por esta misma empresa, que un 31% de las cargas de trabajo ya se realizaban en ese momento en la nube pública. La **nube pública** es la forma más común de implementar la informática en la nube, donde la infraestructura pertenece y es administrada por un proveedor de servicio. Ejemplos de empresas que ofrecen este tipo de nube son *Amazon Web Services* (*AWS)*, *Microsoft Azure o Google Cloud*, de los cuales hablaremos más tarde y ofrecen buenas opciones para servicios *serverless*, arquitecturas que pretenden cambiar el mundo de la informática con su planteamiento de una infraestructura sin servidor. Un 19% de las cargas de trabajo ya se realizaba en la **nube privada**. La nube privada está compuesta por recursos informáticos que utiliza exclusivamente una empresa u organización, esta puede estar alojada en un CPD u hospeda en un proveedor de servicios externo. Y, por último, un 18% en **nubes híbridas**, las cuales son una combinación de nube privada y nube pública como se muestra en la figura 1.

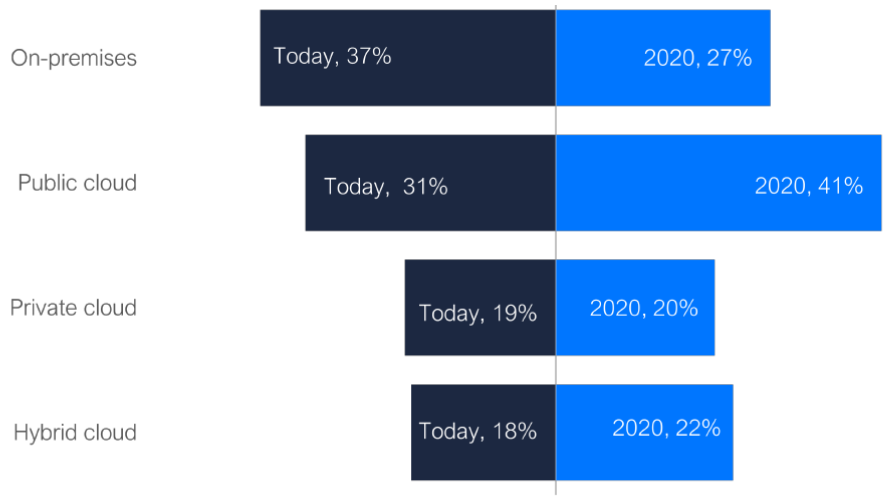



Figura 1. Where will workloads run (Today versus 2020) [12].

Otros datos indicadores del auge de estas tecnologías son que aproximadamente el 30% de todos los presupuestos de las tecnologías de la información (TI) se asignaron a la computación en la nube en 2018, según la revista *Forbes* [10], correspondiendo un 48% a servicios *Software-as-a-Service* (SaaS), un 30% a servicios *Infrastructure-as-a-Service* (IaaS) y un 21% a servicios *Platform-as-a-Service* (PaaS), los cuales se explicarán más adelante.
Y según *Gartner* [13] más de 1.3 trillones de dólares serán gastados por empresas de TI para moverse a la nube en 2020.

Con todos los datos mencionados anteriormente y los grandes números que se manejan, es indudable afirmar que las tecnologías en la nube son parte del presente de la informática y una apuesta segura para el futuro cercano, donde personas influyentes de la industria apuestan por que en el futuro las nubes híbridas cobrarán más fuerza. Aún así, es difícil prever cual es el futuro de la informática con lo rápido que avanza la tecnología y cabe la

posibilidad de que el desarrollo en la computación cuántica, inteligencia artificial y redes neuronales den una nueva vuelta a los sistemas informáticos.

Trabajando con estas tecnologías me crucé con un modelo de arquitectura que había surgido en el núcleo de la computación en la nube, el modelo *serverless* o "Sin servidor". Este modelo destaca frente a modelos más tradicionales por que intenta eliminar toda la administración del lado del servidor, facilitando la escalabilidad y reduciendo costes. Cabe destacar que antes de la existencia de este modelo ya se usaba el término *serverless* en la nube, ya que el uso de ésta eliminaba gran parte de la administración que se tenía que hacer con servidores monolíticos, de ahí el "sin servidor", pero no era un modelo en sí mismo. Realmente, los primeros servicios *FaaS* que hoy en día son la esencia de lo que se conoce como modelo *serveless* nacieron en 2010 a manos de *startups* como *PiCloud.* En cambio, con la salida de *AWS Lambda,* primer servicio *FaaS* de un gran proveedor este modelo cobró fama e importancia y se empezó a desarrollar hasta lo que conocemos hoy en día.
Este modelo me llamó la atención por la peculiaridad de carecer de un servidor, al menos desde el punto de vista del cliente, para ejecutar aplicaciones. Además, en una encuesta publicada por *StackOverflow* se mostraba que era la segunda plataforma más deseada por los desarrolladores tanto en 2017 [14] como en 2018 [15] únicamente por detrás de *Linux*.

Por lo que la principal motivación de embarcarme en la realización de este TFG es el interés en tecnologías tan prometedoras como la nube, así como estudiar un modelo tan innovador e interesante como es *serverless*.

## 1.2 Objetivos

A partir de las motivaciones anteriores, resulta claro y lógico la realización de un documento que explique en detalle que es el modelo *serverless*, para ello será necesario fijar unos objetivos que determinen el alcance de este trabajo.

Mi primer objetivo en este trabajo de fin de grado era **estudiar** qué es el modelo *serverless* y en que se basan estas arquitecturas, ya que en ninguna de las asignaturas del grado lo había estudiado. El segundo objetivo y ligado con el anterior es **formarme** en este campo concreto de las tecnologías en la nube. En tercer lugar, se quiere analizar de forma detallada las **ventajas e inconvenientes** que presentan estas arquitecturas en el contexto de los sistemas en la nube frente a otros modelos arquitectónicos más tradicionales.
En cuarto y último lugar, se quiere realizar un **documento** que pueda servir a cualquier usuario que quiera estudiar y entender en detalle en que se basa el modelo *serverless*, de forma que pueda evitar leer información dispersa y sesgada de los proveedores de servicios en la nube o documentos técnicos poco fiables, y que por ello cuente con información contrastada, bien argumentada, probada y en la medida de lo posible referenciada con otras fuentes fiables. Añadido al último punto, también está el objetivo de que el documento no solo pueda servir a cualquier persona que se quiera formar, sino también a cualquier persona que quiera desarrollar una aplicación o emprender un proyecto usando este tipo de tecnología, de forma que el documento no solo este orientado a explicar en sí el modelo, sino también esté explicado desde el punto de vista del desarrollador.

Para llevar a cabo los objetivos anteriormente expuestos será necesario realizar una **investigación exhaustiva** sobre el modelos *serverless*, contrastar toda información con fuentes fiables y los conocimientos adquiridos durante el grado. Además de desconfiar de

toda información obtenida que pueda extraerse de la documentación de un proveedor de servicio, ya que, al fin y al cabo, es su producto y su opinión está sesgada para obtener un buen margen de ventas.

El objetivo final de este trabajo, además de formarme, es que todo usuario que quiera acudir a este documento termine entendiendo qué es el modelo *serverless* y sus características, comprenda desde un punto de vista del desarrollo las ventajas e inconvenientes de estas arquitecturas y le pueda servir como punto de partida para seguir trabajando e investigando estas tecnologías.

## 1.3 Organización de la memoria

La organización de la memoria se ha estructurado cumpliendo con la normativa de TFG con ayuda de la plantilla proporcionada: introducción, estado del arte, diseño, comprendiendo *serverless,* desarrollo del proyecto, discusión, pruebas y resultados, conclusiones y trabajo a futuro.

En la **introducción** se exponen las principales motivaciones del trabajo causadas principalmente por lo prometedor de la computación en la nube y las arquitecturas *serverless* y se fijan los objetivos que definirán el alcance del trabajo.

En el **estado del arte** se desarrolla el contexto histórico de las tecnologías *cloud* y las arquitecturas *serverless* y como se ha ido desarrollando la idea que ha generado este modelo. Además, se presentan los principales proveedores de tecnologías *cloud* y las diferencias entres sus servicios *Function-as-a-Service* y se explican los diferentes servicios que se proveen en la nube y los diferentes tipos de nube según “The NIST Definition of Cloud Computing” [16].

En el **diseño** se indican los pasos a seguir que se han definido para la realización del trabajo.

En el apartado **comprendiendo *serverless*** se explica de forma más detallada que son las arquitecturas *serverless* y se explican algunas de sus características principales, ayudándose de un par de ejemplos que sirven para la comprender mejor el cambio de planteamiento de arquitecturas tradicionales a las arquitecturas *serverless* y se destancan las ventajas y desventajas que estas suponen en los ejemplos para que el lector de una forma más suave comprenda este modelo.

A continuación el **desarrollo** se exponen las características principales de las arquitecturas *serverless* organizadas por apartados.

En el apartado de **discusión** se discuten algunos temas y se realiza una comparación entre servicios *PaaS,* sistemas de microservicios y arquitecturas *serverless*. Además, la parte más amplia de la discusión comprende el apartado de ventajas e inconvenientes donde se exponen y discuten las ventajas y los inconvenientes de las arquitecturas *serverless.*

En el apartado de **pruebas y resultados** se expone el estado de las pruebas y los resultados del trabajo.

Por último, se exponen nuestras **conclusiones** sobre el TFG y el **trabajo a futuro**, en el se exponen las ideas que permitirían ampliar el trabajo.

# 2 Estado del arte

## 2.1 Contexto histórico

Aunque el concepto de computación en la nube pueda parecer altamente novedoso y sólo en los últimos años parece haberse desarrollado en profundidad, la idea subyacente a este modelo de computación nació en los años cincuenta del siglo pasado a manos de Herb Grosh, que afirmó que "las economías se podrían adaptar mejor si confiaban en el almacenamiento de datos centralizado, y no en equipos individuales", si bien es cierto que esta frase no solo puede a referirse a computación en la nube sólo, ya que habla de almacenamiento de datos centralizado y esto puede referirse a sistemas tradicionales que se encuentren en un CPD.
Esta idea continuó evolucionando en los años sesenta a manos de J.C.R Licklider y John McCarthy que introdujeron ideas como "tecnología de tiempo compartido" o "red galáctica", siguió desarrollándose a manos de Douglas Parkhill en 1996 con su libro "El desafío de la utilidad de la computadora" en el que exploró a fondo muchas de las características actuales de la computación en la nube como el aprovisionamiento elástico a través de un servicio de utilidad.
Uno de los pioneros en la computación en la nube fue *Salesforce* que en 1999 introdujo el concepto de entrega de aplicaciones empresariales a través de páginas web. Finalmente, en la década de los 2000 ya empezaron a nacer estos servicios a manos de *AWS* en 2002 y *Google Cloud* en 2006 entre otros.

Los servicios *FaaS* y por lo tanto lo que entendemos hoy como la computación *serverless* no empezó a surgir hasta 2010 donde algunas *startups* como *Picloud,* ofrecieron estos servicios. Pero cuando cobró importancia fue con la salida de *AWS Lambda* a manos de *AWS* en noviembre de 2014 y a raíz de este hecho el resto de los proveedores *cloud* desarrollaron también sus servicios *FaaS*. Los servicios *FaaS* siguieron desarrollándose hasta que entre los años 2016 y 2017 cobraron gran importancia. Dado lo novedoso de estas arquitecturas surgio gran cantidad documentación y fuentes pero no todas estas eran fiables por lo que para el desarrollo de este trabajo las **principales fuentes** de consulta que se han utilizado han sido las diferentes lecturas de Mike Roberts, experto en arquitecturas *serverless*. Este autor cuenta con numerosas ponencias publicadas, informes técnicos como "Serverless Architectures" [1] o también "Learning Lambda" [2] el cual es el primero de una serie de informes técnicos que explican las arquitecturas *serverless* usando *AWS Lambda*. De este autor junto con John Chapin, también me ha sido de gran ayuda "What is Serverless?" [3]. Otras de mis principales fuentes han sido los informes técnicos realizados por el BBVA [4,5]. Los artículos publicados por el *IEEE "*A preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms", [6] y "Serverless Computing: Design, Implementation, and Performance", [7]. Y por último, la lectura "Severless Architectures on AWS", [8]. Ha habido más artículos e informes técnicos destacables, pero estos trataban de características concretas de las arquitecturas *serverless* por lo que serán citados en el momento que se describan estas características.

## 2.2 Servicios en la nube

Con el nacimiento y el desarrollo de los servicios en la nube nacen diversos tipos de servicios que se pueden obtener a través de estos proveedores, concretamente las arquitecturas *serverless* nacen a manos de *AWS Lambda* en 2014.

### 2.2.1 Tipos de servicios en la nube

Para comprender el significado de las arquitecturas *serverless* es necesario repasar los diferentes servicios que proporcionan las tecnologías en la nube y su motivación, de esta forma podemos ubicar las arquitecturas *serverless* en los servicios de los proveedores *cloud*.

- ***IaaS*** (***Infrastructure as a service***): Este servicio provee de servidores, almacenamiento, redes de comunicaciones entre otros muchos servicios, que podemos gestionar fácilmente y a nuestro gusto a través de una web o vía *API*. Un ejemplo de este servicio es *AWS EC2* (*Elastic Compute Cloud*), que provee de capacidad de computación informática segura y escalable, es decir puedes hacer uso de instancias remotas, seguras y escalables de forma sencilla.

- ***PaaS*** (***Platform as a service***): Este servicio provee de una plataforma en la que desarrollar, desplegar, ejecutar y administrar nuestro software, de esta forma podemos ahorrar costes a la hora de construir y mantener servicios como tenemos que hacer con *IaaS*.
  Un ejemplo de servicio *PaaS* es *Heroku*, que permite a los desarrolladores construir, ejecutar y operar con sus aplicaciones.

- ***SaaS*** (***Software as a service***): Este servicio es el más común, los usuarios finales no se tienes que preocupar ni de su desarrollo ni del despliegue, simplemente el usuario final hace uso del servicio y el proveedor se encarga del mantenimiento, operación y soporte del producto.
  Un ejemplo de servicios *SaaS* es *G Suite*, en el que *Google* provee de un paquete integrado de aplicaciones de colaboración y productivas seguras.

- ***BaaS*** (***Backend as a service***): Los servicios *back end* comprenden aplicaciones de terceros que sirven para administrar la lógica y el estado del servidor. Estos sistemas son más complejos y usan entornos con numerosas bases de datos y servicios de autenticación que dependen de la nube. La diferencia de un servicio *BaaS* con un servicio *SaaS* es que este último se le provee a un usuario final, en cambio el servicio *BaaS* se le provee al desarrollador para que pueda centrarse en el desarrollo, estos servicios proveen a la aplicación de servicios *cloud* tal como almacenamiento en la nube, herramientas analíticas y de administración.
  Un ejemplo de servicio *BaaS* es *Firebase*, el cual es un producto de *Google* orientado a aplicaciones móviles que provee de almacenamiento, gestión de usuarios y herramientas para el desarrollo de aplicaciones web o móviles.

- ***FaaS*** (***Function as a service***): Es un tipo de servicio *cloud* que proporciona una plataforma que permite a los desarrolladores desarrollar, ejecutar y administrar una funcionalidad sin necesidad de crear ni mantener una infraestructura.
  Algunos ejemplos de servicios *FaaS* son *AWS Lambda*, *Google Functions* o *Azure Functions*, estos servicios permiten ejecutar funciones sin administrar ninguna

infraestructura.
Cabe destacar que las arquitecturas *serverless* no podría existir sin los servicios *FaaS*, aunque tiene algunos matices que lo llegan a diferenciar.

Estos dos últimos tipos de servicios tienen especial importancia para mi TFG ya que las arquitecturas *serverless* se componen de ambos como se explicará posteriormente.

La siguiente imagen ilustra de forma esquemática los principales servicios *cloud* definidos en el "The NIST definition of cloud computing" [16], que son *IaaS, SaaS* y *PaaS.*

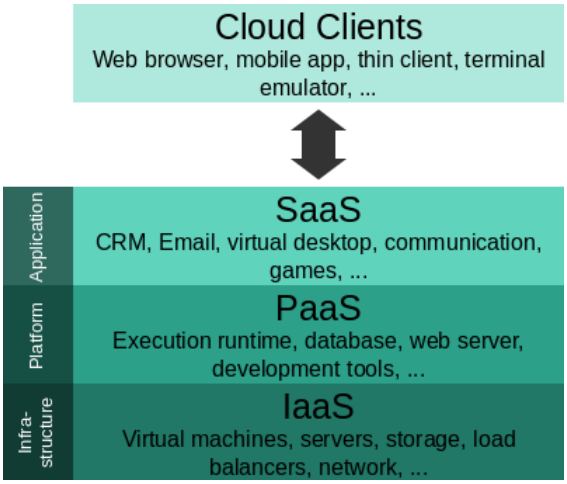



Figura 2. Capas de la computación en la nube. [17]

Pero como se ha indicado antes *BaaS* y *FaaS* van a ser los servicios más importantes relacionados con las arquitecturas *serverless* y no están ubicados en la anterior figura ni definidos en "The NIST definition of cloud computing", por lo que ya definidos se va a discutir de donde ubicar estos servicios.

En el caso de *BaaS* es un servicio que provee de servicios *back end* a un *front end* como por ejemplo una base de datos, un servicio de autenticación o un servicio de colas. Se podría ubicar en la siguiente imagen entre las capas *IaaS* y *PaaS.* En primer lugar, no llegaría a nivel de plataforma, es decir de los servicios *PaaS* ya que estos servicios no llegan a proveer una plataforma donde el desarrollador pueda desarrollar, desplegar, ejecutar y administrar su software como hacen estos servicios. En segundo lugar, se acerca mucho a los servicios *IaaS* y pueden incluso llegar a considerarse de este tipo, ya que en cierto modo también puede proveer servicios de infraestructura, pero están más orientado para un *front end.* Un ejemplo que se sitúa en un caso límite, un almacenamiento de objetos como *AWS S3 (Simple Storage Service)* se considera un servicio *IaaS*, pero en cambio *Firebase* que es una plataforma que provee un conjunto de herramientas como la de base de datos, se considera *BaaS*, ya que está orientado a trabajar con todas sus herramientas directamente con un *front end* de forma sencilla.

Los servicios *FaaS* se encuentran entre las capas *PaaS* y *SaaS.* En primer lugar, no está al nivel de los servicios *PaaS,* ya que tiene un nivel de automatización mayor, por otro lado, en los servicios *PaaS* todavía hay que administrar parte de las herramientas *back end* que provee para el funcionamiento de la aplicación, mientras que en los servicios *FaaS* no tienes por qué ya que se comunica con otros servicios *back end* a través de llamadas a una *API*. Por último, no llega a la capa de *SaaS* debido a que en los servicios *FaaS* hay

que desarrollar y administrar el cliente. En la siguiente figura [18] se muestra donde se sitúa *FaaS* entre los servicios definidos en “The NIST definition of cloud computing”.

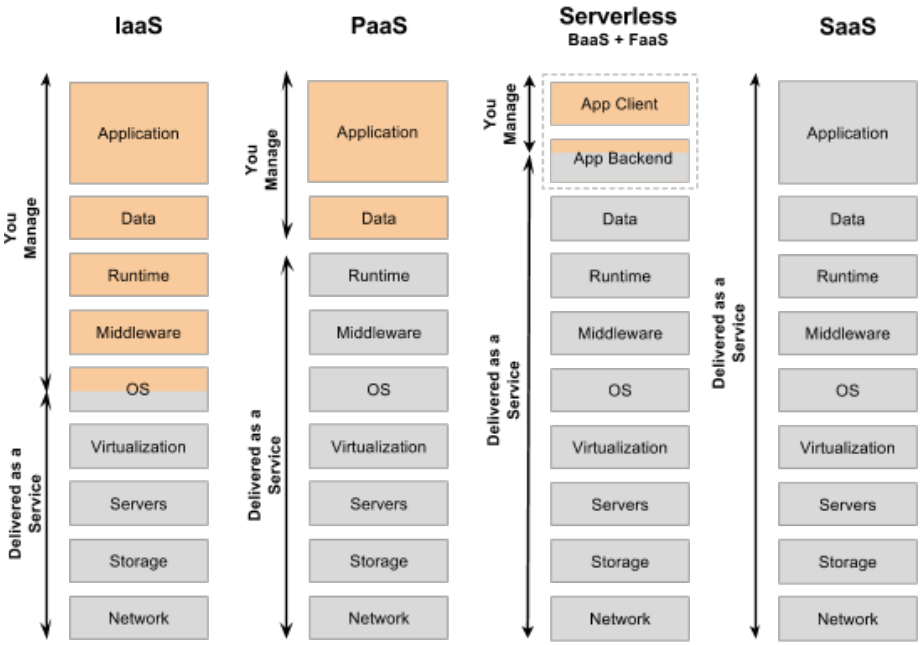



Figura 3. Grado de automatización usando serverless. [18]

### 2.2.2 Proveedores de servicio, nube pública, nube privada, nube híbrida y nube comunitaria

Los **principales proveedores** de servicios de **nube pública** que además ofrecen sus servicios *FaaS* son *AWS* con *AWS Lambda, Google Cloud,* la cual es la plataforma de nube donde *Google* ofrece sus servicios *cloud* y cuenta con *Google Cloud Functions* como servicio *FaaS* y *Microsoft Azure,* la cual es la plataforma de servicios en la nube de *Microsoft* y cuenta con *Azure functions* como servicio *FaaS.*

Cabe destacar que me veo obligado a centrar todo el trabajo en un proveedor de servicio *cloud*, ya que hay muchos proveedores de servicios *cloud* en el mercado y no me es posible cubrirlos todos con todas sus especificaciones. Por lo tanto, el TFG va a basarse en el proveedor de servicios *AWS*, ya que es el líder indiscutible en cuanto a servicios *cloud*, así lo dice *Gartner* en su estudio anual que plasma en el conocido “Gartner Magic Quadrant” [19], además fue la primera plataforma en lanzar esta idea y la que más funcionalidades incorpora.


Figura 4. Magic Quadrant for Cloud Infrastructure as a Service Worldwide. [19]

En la siguiente figura tomada de un artículo del banco *BBVA* sobre *serverless* [4] se muestran las principales diferencias entre los servicios *serverless* que ofrecen los principales proveedores de servicios en la nube y sus características.

| | AWS Lambda | Google Cloud Functions | Azure Functions |
|---|---|---|---|
| Supported Languages | JavaScript (NodeJS), Python, binary | JavaScript (NodeJS) | C#, F#, Node.js, Python, PHP, batch, bash, binary |
| Event triggers | HTTP, tiempo, S3, Kinesis, Streams, SNS, SES, Cognito, Cloud Formation, Cloud Watch, Code Commit, Config, Echo, Alexa, API Gateway | HTTP, tiempo, Google Cloud Storage, Gmail push | HTTP, QueueTrigger, Azure Storage, Events Hubs, QUeues, Tables, Notification Hub, DocumentDB, Notification hub, Twilio |
| Persistency | Via ENV (stateless by default) | n/a | Yes, via ENV |
| OS | Linux x64 | Linux x64 | Windows 32-vbit |
| License | Closed | Closed | Open |
| Max execution time | 15 minutes | Unlimited | Unlimited |
| Mac functions per project | Unlimited | Unlimited | 20 |
| Max current functions | 100 | Unlimited | Unlimited |
| Reserved sources | Memory reserved per function | n/a | Memory reserved per service |

Tabla 2. Caracterísiticas de las tecnologías *FaaS* de los principales proveedores de nube pública. [4]

El caso más común para desarrollar un proyecto basado en arquitecturas *serverless* es usar alguno de los principales proveedores de la **nube pública**. Aunque ya se definió

brevemente en el apartado de motivación voy a explicar tal y como se definión en “The NIST Definition of Cloud Computing” [16] los diferentes modelos de despliegue en la nube.

“La **nube pública** provee infraestructura de uso abierto para el público general. Debe de ser propiedad, administrado y operado por un negocio, institución académica, organización del gobierno o una combinación de las anteriores.” [16]

En el caso de *AWS Lambda* se sabe poco sobre cómo se ejecutan las funciones, en la documentación de *AWS* figura que usa un sistema de contenedores, pero no se sabe nada más allá. En el caso de *Azure* tenemos más detalles, las funciones se ejecutan sobre *Azure WebJobs* y éstos tienen publicado su entorno de ejecución en *GitHub.* Por último, en el caso de *Google Functions* las funciones se ejecutan sobre entornos nativos de *Node.js.*

“La **nube comunitaria** es la infraestructura que se provee para uso exclusivo de un grupo de consumidores que comparten preocupaciones, por ejemplo, requisitos de seguridad, política, etc. Debe de ser propiedad, administrada y operada por una o más organizaciones de la comunidad, un proveedor externo o una mezcla de los anteriores.” - [16]

Cuando se habla de **nube privada** se refiere a ejecutar *on-premise*, es decir, en el entorno de la propia empresa o persona estas funciones, para ello hay multitud de servicios para llevar acabo esto, me voy a centrar en algunos de los más usados que también se nombran en el artículo nombrado anteriormente del BBVA [4] que son *Iron Functions*, *OpenStack Picasso*, *Fission*.

En el caso de *Iron Functions* esta es una aplicación que provee una plataforma de computación *cloud open source* que se puede utilizar con cualquier tipo de nube, privada, pública o híbrida y puede ser desplegada como una aplicación u ofrecida como un servicio *PaaS* o *IaaS*. En cuanto al servicio FaaS que ofrece este se despliega sobre servidores físicos.

*Fission* es un *Framework* de funciones *serverless* sobre *Kubernetes* nativo, esto permite unir de forma sencilla sistemas de microservicios tradicionales con los implementados en *serverless*, de forma que todo esté integrado en *Kubernetes* permitiendo así beneficiarnos de algunas funcionalidades de estos servicios como la monitorización, agregación de *logs* y publicación de servicios.

*OpenStack Picasso* provee una *API* para servicios *FaaS*. Esta se puede utilizar con las dos aplicaciones mencionadas anteriormente y se puede integrar con servicios de *OpenStack* sin este tener un control sobre la infraestructura que ejecuta las funciones, de forma que tan solo permite crear funciones privadas que son visibles para los usuarios de un *tenant* de *Openstack*.

Por último, la **nube híbrida** es una combinación de dos o más de los anteriores modelos de despliegue en la nube. Lo más común es la combinación de nube pública y nube híbrida.

# 3 Diseño

En el diseño se **indican** los pasos a seguir para la realización del trabajo**,** ya que este es un estudio sobre las arquitecturas *serverless*.

En primer lugar, para la realización del trabajo se va a escoger la bibliografía, contando principalmente con lecturas sobre las arquitecturas *serverless*, documentación oficial de los principales proveedores de servicios *FaaS,* informes técnicos y artículos sobre computación en la nube y arquitecturas *serverless.*

En segundo lugar, se va a realizar una lectura exhaustiva de la bibliografía, descartando toda lectura que parezca estar muy sesgada o contenga información errónea. Además, se va a leer desde un punto de vista agnóstico toda la bibliografía, haciendo especial énfasis en los informes técnicos y documentación oficial ya que estos pueden contener errores o tener preferencias por las arquitecturas a estudiar.

En tercer lugar, de la bibliografía restante se va a extraer de toda información que se considere relevante para la realización del trabajo, se va a contrastar y en la medida de lo posible se van a realizar pruebas de concepto para probar o desmentir la información.

Por último, se va a plasmar todo lo documentado en el paso anterior en el trabajo, desarrollando y ordenando su contenido para que este sea lo más claro posible, además, se realizarán aportaciones personales formadas durante el estudio y trabajo con estas arquitecturas, y se discutirán gran parte de la información documentada para obtener así un documento cumpla los objetivos planteados.

# 4 Comprendiendo *serverless*

## 4.1 ¿Qué es *serverless*?

Las arquitecturas *serverless* son un modelo arquitectónico en el que los servidores (físicos o en la nube) dejan de existir para el desarrollador y el código corre en "ambientes de ejecución" sin estado (*stateless*) y efímeros que, siguiendo el esquema de la computación en la nube, habitualmente administran proveedores *cloud* como *Amazon* con *AWS Lambda*, *Azure* con *Azure Functions*, *Google* con *Cloud Functions*, *IBM*, *Alibaba* etc.

La esencia reside en que el proveedor soporta ciertos lenguajes, de forma que el programador inserte su código en una función pública (*Función Lambda*) y el proveedor sea el que se encargue de administrar toda la infraestructura.
Esto permite el escalado horizontal y vertical, además, una de las características más destacadas es que solo se paga por el uso que se haga de la misma, a diferencia de tener un servidor propio dando servicio constantemente (que sería el caso de uso normal) por lo que se ahorra todo el tiempo ocioso de la máquina, ya que se cobra por ejecución. Normalmente, se busca que cada una de estas funciones tenga una funcionalidad muy concreta.

Cabe destacar que durante la realización de este trabajo me he encontrado que se refiere a *serverless* como *FaaS*, otras veces como *BaaS* y otras siendo estos dos últimos dos subtipos diferenciados de estas arquitecturas.

La mayoría de las veces, en la bibliografía se habla de *serverless* como *FaaS* sin incidir en mucho más detalle. Aunque esto puede ser correcto en determinadas circunstancias, para ser estricto se debe tener en cuenta que las arquitecturas *serverless* realmente se componen de servicios *FaaS* y *BaaS*. Se le suele denominar un servicio *FaaS* porque el núcleo de *serverless* es la funcionalidad, es decir, las funciones, pero si fuese sólo así tendría muchas limitaciones, en cambio, combinando *FaaS* y *BaaS* es capaz de abarcar un espectro mucho más amplio. Además, en esencia los servicios *FaaS* son *serverless* porque el desarrollador no se tiene que preocupar de ninguna infraestructura.
Como he dicho anteriormente, el componente *FaaS* es la funcionalidad principal, la ejecución de la lógica programada sin la preocupación de la infraestructura básica como el entorno de ejecución o el almacenamiento.
El componente *BaaS* se encuentra en la infraestructura que se puede manejar a través de una función con servicios externos o otros servicios proporcionados por el propio proveedor dado el alto acoplamiento, tales como la autenticación, almacenamiento en bases de datos relacionales y no relacionales… etc, y todo ello realizado desde el código.

### 4.1.1 Comprendiendo *serverless*, ejemplo 1

Un ejemplo para comprender lo relatado hasta ahora y ver la comparación de una arquitectura tradicional y más simple frente a una arquitectura *serverless*, es una aplicación de tipo "*User Interface driven applications*" como por ejemplo una tienda *online*.

En una aplicación de tipo tienda típica con una arquitectura WWW típica tendríamos un cliente ligero con un componente *HTML* y *Javascript* ejecutándose en un navegador, por ejemplo. En la parte del servidor tendríamos toda la lógica de la aplicación como la autenticación, navegación entre páginas, búsquedas y transacciones desarrollados con un lenguaje como *Python*, *java* o *C* que además se comunica con una base de datos relacional de tipo *Postgresql*, por ejemplo.

De forma que un esquema tradicional sigue una arquitectura en tres capas que se puede repartir en dos o más niveles:

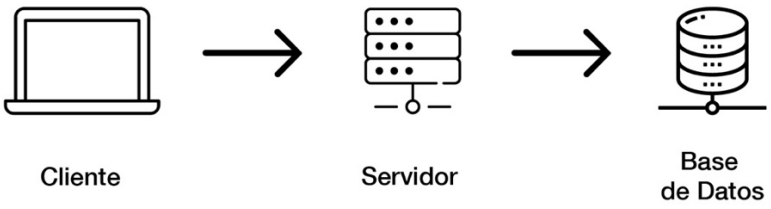



Figura 5. Arquitectura tradicional simple de una aplicación de tipo tienda.

En el diagrama anterior, dado que son tres componentes cada uno tiene una alta criticidad, sobre todo el servidor que es el que alberga toda la lógica. Si hubiese algún problema en el servidor como perderíamos todo el servicio de la tienda online, esto es debido a que en general en esta arquitectura existe un alto acoplamiento entre el servidor y el resto de los componentes.

En una arquitectura *serverless,* el cliente se mantendría similar pero la arquitectura tendría toda la lógica de forma más distribuida de forma que tendríamos un conjunto de bases de datos de fácil administración y con alta disponibilidad, un servicio de autenticación seguro, un *API Gateway* para administrar fácilmente las diferentes transacciones, y para las funciones básicas de la tienda como la búsqueda y la compra en funciones lambda (servicio *FaaS* de *AWS*).

El diagrama del ejemplo de tienda *online* en una arquitectura *serverless*, sería así:

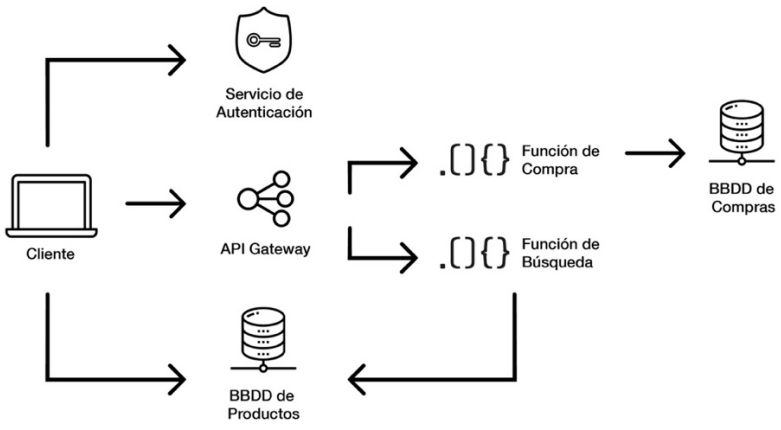



Figura 6. Arquitectura *serverless* de una aplicación de tipo tienda.

Esta arquitectura cumpliría un bajo acoplamiento dado que cada componente es independiente, pero hay una alta cohesión porque estos componentes están muy relacionados entre sí, de forma que si fallase algún componente no tendría por qué producirse una pérdida de servicio, simplemente se vería afectado el funcionamiento de la aplicación. Además, la gran ventaja de esta arquitectura es que al estar en un proveedor como *AWS* todo es fácilmente administrable, cumple unos requisitos de seguridad, disponibilidad, escalabilidad y fiabilidad sin tanto esfuerzo extra como en una arquitectura tradicional, y si todos los elementos son del proveedor nos aseguramos de una óptima interacción entre componentes.

Podemos concluir el ejemplo remarcando el hecho de que en una arquitectura tradicional toda la lógica, control y seguridad se administran desde el servidor a diferencia de en *serverless* donde cada elemento realiza su propio rol más independiente, aunque no sea la única arquitectura que provee estas ventajas.

Esta idea de una arquitectura formada por componentes más independientes se acerca mucho al enfoque de los microservicios, este enfoque otorga una mayor flexibilidad y facilidad de cambio a través de actualizaciones independientes de los componentes. Por el contrario, necesitamos una mejor monitorización y más distribuida, confiamos en la seguridad que nos proporciona la plataforma y añadimos una mayor complejidad debido a que está compuesto por múltiples componentes de *back end*.
Por lo tanto, la decisión de usar una arquitectura *serverless* depende del contexto y del problema que queramos abordar.

### 4.1.2 Comprendiendo *serverless*, ejemplo 2

Otro ejemplo diferente para comprender cómo funcionan las arquitecturas *serverless* y qué ventajas presentan es un servicio de procesamiento *back end*, como por ejemplo un sistema de publicidad online.

Este sistema necesita una respuesta muy rápida a cualquier interacción del usuario y capturar toda la actividad que lleve a cabo del usuario para procesarla más tarde. En concreto, se necesita que, si el usuario pincha en un anuncio, rápidamente se le redireccione a la página deseada y al mismo tiempo registrar esa información para que el sistema de publicidad lo guarde y pueda proveer unas sugerencias más afines a los gustos del usuario.
Para ello, necesitamos que una vez se registre el evento de pinchar en el anuncio se le redireccione, y de forma síncrona se registre un mensaje en una cola de mensajes que será procesado de forma asíncrona por un procesador y lo registrará en una base de datos.

El diagrama del ejemplo en una arquitectura tradicional:

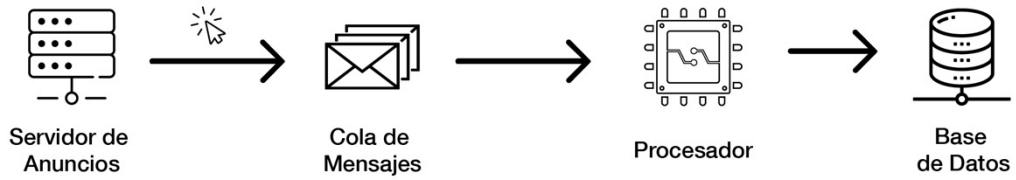



Figura 7. Diagrama de arquitectura tradicional de un sistema de sugerencias.

En diagrama del ejemplo una arquitectura *serverless*:

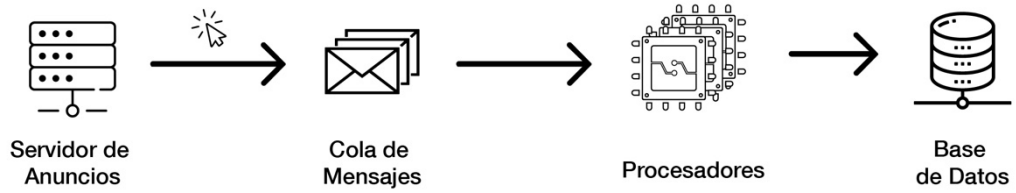



Figura 8.Diagrama de arquitectura *serverless* de un sistema de sugerencias.

En el ejemplo anterior se puede apreciar que la diferencia entre las dos arquitecturas es mínima. Esta diferencia reside en que usando una arquitectura *serverless* se crean copias paralelas de la función para procesar los mensajes, esto es mucho más eficiente y escalable, ya que hemos sustituido un sistema de tipo consumidor y productor por una función asíncrona, lo cual es un uso común de las tecnologías *serverless*.

# 5 Desarrollo

## 5.1 Propiedades de las arquitecturas *serverless* en detalle

Basándome en la definición proporcionada por *AWS* de qué es *AWS Lambda* voy a tratar en detalle las propiedades que caracterizan las arquitecturas *serverless*.

Fundamentalmente, *serverless* es correr una aplicación sólo mediante el código sin necesidad de administrar ningún servidor ni *back end*. Además, se ayudan de tecnologías modernas como contenedores o *PaaS*.

Uno de los puntos fuertes de usar *FaaS* es que los *frameworks* y las librerías ya no son limitantes, porque no hay que preocuparse de ellas. En general, los servicios *FaaS* soportan numerosos lenguajes como por ejemplo *Javascript*, *python*, *Go*, *Java*, *Clojure*, *Scala*, etc. Pero realmente puede ejecutar cualquier lenguaje que pueda ser compilado con *Unix* porque estos servicios permiten ejecutar cualquier proceso que esté ligado con un artefacto de despliegue, es decir, que tenga una imagen generada, esto último lo explicaré en detalle más adelante.

Por contra, elegir la opción *FaaS* también requiere adaptarse a ello, ya que no conserva el estado y hay un tiempo limitado de ejecución, por lo que en el ejemplo del sistema de publicidad el cambio comprometería al método principal que llama a la funcionalidad, ya que este se ejecutaría en la función *lambda* y la interacción con la cola de mensajes, pero el resto se mantendría.

Para el despliegue del servicio las arquitecturas *serverless* se diferencian mucho de las tradicionales ya que bastaría con subir la función que queremos que ejecute el servicio, al proveedor, y entonces este se encargaría de administrar el resto de los recursos (procesos, máquinas virtuales, etc), además se puede incluir este despliegue en el *pipeline* de *Continuous-integration-continuous-deployment* (*CICD)* para agilizar el proceso.

El escalado horizontal es automático y realizado por el proveedor, como se explicaba anteriormente no hay que preocuparse por la administración, y por lo tanto al no haber estado en la aplicación el proveedor crea y destruye el número de tareas de un servicio a su gusto. Es decir, si por ejemplo en nuestro sistema de publicidad en un momento dado hay mucha gente pinchando en los anuncios, nuestro sistema podrá aguantarlo ya que el proveedor paraleliza todas las tareas que sean necesarias para que el servicio funcione correctamente.

El uso habitual de las funciones o servicios *FaaS* es que se ejecuten tras un evento, como por ejemplo un cambio en un *S3*, la llegada de un mensaje con el servicio *AWS Kinesis*, un proceso planificado o tras una consulta HTTP. Esta última se realiza a través de un *API Gateway* como en el ejemplo de la tienda para comprar o buscar productos.

### 5.1.1 Estado de la aplicación

Un concepto fundamental para tener en cuenta y que condiciona todo proyecto que se quiera llevar a cabo es el hecho de que las arquitecturas *serverless* no conservan el estado, y esto puede ser un gran inconveniente. A pesar de que en el contenedor o en la máquina haya almacenamiento de sobra, las variables y todo el almacenamiento temporal no tiene por qué guardarse y sería erróneo suponer que se va a guardar, por este motivo se suele decir que las arquitecturas *serverless* son arquitecturas sin estado o *stateless*. Esto en algunos casos no tiene por qué suponer un problema, ya que existen tecnologías y metodologías que no mantienen el estado y se desarrolla en ellas, como por ejemplo *The twelve-Factor App*, la cual es una metodología para construir aplicaciones *SaaS* de este modo.

En cualquier caso, el hecho de que no guarde el estado puede llegar a ser limitante y es condicionante a la hora de empezar un proyecto con arquitecturas *serverless*.

Destacar, que si fuese necesario existen vías para poder almacenar el estado como por ejemplo guardar los datos en bases de datos, ya bien sean externas, o del proveedor, en cachés como *Redis* y incluso es posible guardar objetos en *buckets* con ayuda de un *S3*. A pesar de que existan estas opciones, no son tan recomendables y si estás pensando en guardar el estado de una de estas formas usando una arquitectura *serverless* es posible que esta arquitectura no sea la solución óptima.

### 5.1.2 Duración de la ejecución

Es interesante y necesario saber qué limitaciones se sufren al trabajar con arquitecturas *serverless* en cuanto a tiempo de ejecución se refiere, ya que esto puede limitar algunos casos de uso.

Como máximo las funciones *lambda* tienen quince minutos para ejecutarse, este valor se puede ajustar a un valor menor manualmente si se desea. Antes el límite máximo eran cinco minutos, pero se amplió para que los servicios *FaaS* puedan servir de solución a análisis de *big data*, tratamiento de datos pesados, procesamiento, computación y computación estadística.

Por lo tanto, las arquitecturas *serverless* no están pensadas para programas de larga vida como por ejemplo un demonio (*daemon*), y si se quisiese llevar a cabo sería altamente complejo, ya que habría que crear diferentes funciones que se coordinen entre ellas e incluso se comuniquen para no exceder el tiempo máximo de ejecución y que de alguna forma imitasen el flujo de un programa de larga duración.

### 5.1.3 Latencia de arranque, arranques en frío y arranques en caliente

Otra característica de estas arquitecturas es que al usar servicios *FaaS,* las funciones pueden sufrir cierta latencia al activarse para un evento, denominado comúnmente como latencia de arranque, debido a que son contenedores que se ejecutan bajo demanda, esto, si sucede se denomina arranque en frío. Sí después de activarse, vuelve a haber otro evento y el servicio está disponible, no sufrirá latencia, a esto se le denomina arranque en caliente.

Es necesario tener en cuenta cómo funciona, cuándo se produce y cómo se puede controlar si se quiere aprovechar al máximo el rendimiento de la aplicación.

La latencia de arranque se da porque al utilizar un servicio *FaaS,* el proveedor necesita arrancar una instancia y crear un contenedor con un contexto en función de la configuración que se le haya aplicado a la función, este periodo de tiempo se denomina contexto de ejecución y en él, el proveedor arranca un contenedor con algunas de las configuraciones proporcionadas, además, inicializa las dependencias externas, conexiones a bases de datos y puntos de enlace *HTTP* si los hubiese. Esta latencia puede durar desde unos milisegundos hasta varios segundos dependiendo de si reutiliza una instancia con un contexto ya inicializado y preparado de una ejecución anterior, o tiene que volver a inicializarlo.

Un arranque en frío depende de varios factores para que se dé, el lenguaje de programación que se esté utilizando, el número de librerías cargadas, el proveedor del servicio *FaaS*, cuánto código haya que ejecutar y el entorno de la función como conexiones a Amazon *VPCs*, subredes, bases de datos, puntos de enlace, etc. La parte positiva es que muchos de estos aspectos están bajo el control del desarrollador.

Otro factor importante de estos arranques es la frecuencia de arranque. La frecuencia de arranque es el número de veces que la función se ejecuta en un período de tiempo y cuanto mayor sea la frecuencia menos probabilidades hay de que se produzca un arranque en frío. Por lo tanto, si tenemos una función que se ejecuta tras un evento y suceden 10 eventos por segundo, cada uno de ellos con una duración de 60 ms, es probable que no se produzca un arranque en frío, y por lo tanto, se produzca un arranque en caliente, ya que al proveedor no le ha dado tiempo a retirar la aplicación en lo que se producen dos ejecuciones . Por contra, si la función se ejecuta cada hora, lo más probable es que siempre se produzca un arranque en frío, y por lo tanto tenga una latencia de arranque y un retraso en la respuesta.
Entendiendo esto se puede tener en cuenta el impacto que pueden sufrir los arranques en frío en la aplicación y valorar si se quiere evitar esta situación usando *Keep Alives*.

Los *keep alives* en este contexto se refiere a eventos ficticios en los que la función se ejecuta sin ningún fin más que la de que el proveedor no retire la aplicación de la *lambda* por poco uso y por lo tanto evitar un arranque en frío en el siguiente evento.

Llegados a este punto quizás nos rodea la pregunta ¿Los arranque en frío importan? la respuesta, como siempre en estos casos es: depende, se debe valorar si la latencia que produce en un arranque en frío tiene un gran impacto en la aplicación, y, por lo tanto, si supone un problema. Si supusiera un problema se debería valorar el uso de *keep alives* para compensarlo y si no, quizás las arquitecturas *serverless* no son la solución adecuada para tu aplicación. Cabe destacar que este tipo de servicios avanzan muy rápidamente y es posible que de aquí a unos meses si puedan satisfacer el problema inicial, ya que los proveedores *cloud* no paran de mejorar este tipo de servicios.

# 6 Discusión

## 6.1 API Gateway

Un elemento *BaaS* que se suele utilizar mucho con servicios *FaaS* formando arquitecturas *serverless* para algunas soluciones como en el primer ejemplo de la tienda virtual son los *API Gateway*, a continuación, voy a explicar que es un *API Gateway* y por qué es tan útil en soluciones *serverless*.

Un *API Gateway* es un servidor *HTTP* donde las rutas y los *endpoints* están definidos en su configuración y cada una de estas rutas está asociada a un recurso que maneja ese evento. En nuestro caso, orientado a *serverless* ese recurso estaría asociado a una función *FaaS*, esto permite una gran interacción entre *API Gateway* y la función, lo que puede ser útil en muchos casos.

El funcionamiento de un *API Gateway* ante una consulta sería el siguiente, el *API Gateway* recibe una consulta, busca la ruta y una vez encontrada ejecuta la función correspondiente con el contenido de la consulta original. Una ventaja de usar este servicio es que simplifica los parámetros de entrada para que sean más fáciles de manejar en la función, además ofrece la posibilidad de que se reciba como objeto *JSON*. Una vez ejecutada la lógica de la función el *API Gateway* devuelve el resultado como respuesta *HTTP*.
El uso de un *API Gateway* en arquitecturas *serverless* suele ser muy útil, ya que permite controlar y orquestar todos los eventos con las funciones de la arquitectura *serverless* empleada, simplificando los parámetros de entrada y haciéndolos más manejables, proporcionando mejoras en la autenticación, validación de la entrada, un sistema sencillo de respuestas *HTTP* y más.

Desde mi punto de vista, el uso de este tipo de infraestructura en las arquitecturas *serverless,* siempre que sea necesario, es una ventaja, ya que proporciona muchas mejoras, facilitando y mejorando la gestión de la entrada de nuestro sistema, permitiendo organizar y orquestar las funciones del sistema sin una administración muy avanzada y dar respuestas de forma sencilla.

## 6.2 Comparación entre PaaS y FaaS

En algunos casos podemos encontrar algunas similitudes entre *PaaS* y algunas arquitecturas *serverless*, citando a Adrian Cockcroft, vicepresidente de estrategia de arquitectura *cloud* de *AWS*, “Si tu servicio *PaaS* puede de forma eficiente arrancar instancias en 20 ms que corren medio segundo, entonces se llama *serverless*” [20].

Está claro que en esta frase Adrian está haciendo una comparación muy a alto nivel, porque como se ha expuesto anteriormente en el trabajo, hay muchos matices, ventajas y desventajas que diferencian a aplicaciones construidas mediante servicios *PaaS* de las de arquitecturas *serverless*, pero a lo que se refiere Adrian es que la mayoría de las aplicaciones *PaaS* no están preparadas para estar parándose y levantándose continuamente como respuestas a eventos.

Una ventaja clara de los servicios *FaaS* frente a los servicios *PaaS* es la escalabilidad, en los servicios *FaaS* es algo de lo que no hay porque preocuparse ya que se realiza

automáticamente, en cambio en los servicios *PaaS,* aunque cuenten con auto escalado no llegan a escalar tan a bajo nivel como realizan los servicios *FaaS,* que escalan en función del número de consultas recibidas lo que permite optimizar más los costes.

Sin embargo, los servicios *PaaS* tienen su punto fuerte en el conjunto de herramientas que abarca las cuales pueden cubrir casi cualquier solución que se requiera a nivel de arquitectura, ya que, cuentan con herramientas de diseño y flujo de trabajo, *APIs* completas, administración de la plataforma, control de acceso a la plataforma, entornos de desarrollo, monitorización y notificación de errores y más.

Implementar una solución *PaaS* antes que una *serverless* o contenedores depende del aplicativo, desde mi punto de vista, es una mejora de cara a una arquitectura más tradicional, construida en un servidor *on premise* pero que es muy complejo o imposible implementarlo en una arquitectura *serverless* o mediante contenedores, ya que es un aplicativo o un conjunto de aplicativos que no se pueden simplificar ni fragmentar en subconjuntos más pequeños, además supone una mejora ya que permite una mayor facilidad de administración y gestión del aplicativo y de la infraestructura frente a una más tradicional sin necesidad de perder control a bajo nivel. Un ejemplo de este tipo de casos suelen ser aplicaciones de terceros que se necesitan o nos gustaría implementar en nuestro sistema, pero este solo se puede usar y mantener en una instancia, probablemente de tipo servidor.

## 6.3 Comparación entre sistemas de microservicios con contenedores y arquitecturas *serverless*

Llegados a este punto, nos puede surgir la duda de que es mejor, si utilizar arquitecturas *serverless* o arquitecturas basadas en contenedores de distinto tipo, como *Docker*, *Kubernetes* o *Mesos*, entre otros, o una arquitectura formada por *serverless* con sistemas de contenedores. Esta pregunta no se trata de ninguna tontería ya que las dos tecnologías basadas en microservicios, donde cada microservicio usa uno o varios contenedores están en auge por todas las ventajas que ofrecen, dado que separan toda la funcionalidad en funcionalidades más pequeñas, cada una de estas funcionalidades se ejecuta en un contenedor y se denomina microservicio, al ser una unidad de la aplicación más pequeña fomenta buenas prácticas  favoreciendo el bajo acoplamiento y la alta cohesión.

Por otro lado, también encontramos plataformas *cloud* de contenedores como es el caso de AWS *ECS (Elastic Container Service)* o AWS *EKS (Elastic Kubernetes Service)*, los cuales permiten usar sistemas de contenedores de forma más sencilla aprovechando las ventajas de los servicios *cloud*.

La comparación entre estos dos sistemas es parecida a la comparación realizada anteriormente entre *PaaS* y *FaaS*. *Serverless* mantiene su punto fuerte en que se evita la administración de los sistemas, tiene auto escalado automático y muy preciso ya que va a nivel de consulta y ligado a su aprovisionamiento automático, lo cual permite que la arquitectura este hecha a medida.

Desde otro punto de vista, los clústeres de contenedores permiten auto escalado y *Kubernetes* ya implementó escalado horizontal hace tiempo. Estos sistemas fomentan buenas prácticas como he resaltado al principio de este apartado, fomentan el bajo acoplamiento y la alta cohesión lo cual minimiza los errores y su impacto en el sistema y además, promueve la buena coordinación entre los microservicios. En cambio, existe una

necesidad de administrar estos sistemas cuando se desarrollan y se prueban, y en caso de que exista alguna incidencia los sistemas *cloud* de contenedores intentan minimizar esta administración para facilitar la operación el máximo posible.

Por todo lo anterior, la decisión entre una arquitectura *serverless* o de contenedores en servicios *cloud* depende del estilo y tipo de aplicación que se quiere desarrollar. Por ejemplo, para una aplicación que responde a un evento y tiene distintas funcionalidades como el del ejemplo 1 (tienda online), la mejor decisión es usar una arquitectura *serverless* antes que usar contenedores. En cambios, para una aplicación que responde a distintos eventos síncronos de diversos puntos de entrada es mejor usar contenedores.
Realmente las dos arquitecturas son muy prometedoras y ya se usan en arquitecturas de aplicaciones reales muy importantes, así que la mejor conclusión a la que puedo llegar es que hay que aprovechar ambas y complementarlas usando una arquitectura híbrida, esto no tiene porqué añadir complejidad si se monta correctamente y haciendo uso de buenas prácticas.
De hecho, existen servicios muy prometedores como puede ser *AWS Fargate*, el servicio *cloud* de contenedores *serverless*, el cual voy a explicar en el siguiente apartado.

## 6.4 AWS Fargate

*AWS Fargate* es una alternativa a todos los servicios mencionados anteriormente que une los sistemas *serverless* y los contenedores, de forma que interactúe con los servicios de contenedores *cloud* de *AWS* (*ECS y EKS*) que hace automático el aprovisionamiento y la administración de instancias en las que corren los contenedores, al igual que *AWS Lambda* se paga por uso y la escalabilidad es automática.
Por hacer una comparación *AWS Lambda* es a *EC2* lo que *Fargate* a *ECS* y *EKS.*

Usar *Fargate* puede ser tan buena solución como todas las anteriores, depende de lo que se quiera implementar, pero puede ser muy buena alternativa a usar solo *ECS* o *EKS*.

## 6.5 Stored Procedure as a service y *FaaS*

He introducido este punto porque durante la realización del trabajo he encontrado que en numerosos artículos centran la importancia de las arquitecturas *serverless* y servicios *FaaS* para el uso de procedimientos almacenados (*Stored procedure as a service)*.
Los procedimientos almacenados son pequeños trozos de código con una funcionalidad muy concreta y que normalmente se relacionan con bases de datos o algún servicio de *back end* (no confundir con los procedimientos almacenados que permiten ejecutar funcionalidad, por ejemplo, SQL, dentro de un gestor de bases de datos).

Es cierto que las arquitecturas *serveless* y los servicios *FaaS* permiten ejecuctar procedimientos almacenados ofreciendo soluciones a algunos problemas/limitaciones que plantean. Sin embargo, éste no sería el único escenario en el que aprovechar el potencial ofrecido por este tipo de arquitecturas. Los procedimientos almacenados en muchas ocasiones obligan a que los procedimientos estén en un lenguaje concreto o específico del proveedor, es complicado realizar pruebas a estos procedimientos y es difícil administrar un control de versiones, en cambio, usando *serverless* se incluyen todos los lenguajes que soporte el proveedor de servicio de funciones y no nos limita a un lenguaje, además las pruebas unitarias son tan sencillas hacerlas como con una aplicación normal y para el control de versiones existen servicios de control de versiones como SAR (*Serverless*

*Application Repository*), el cual es un servicio implementado por *AWS* que pone a nuestra disposición un repositorio de código para *funciones lambda*.

Por lo que podemos concluir que las arquitecturas *serverless* son óptimas para guardar procedimientos almacenados, pero aun así este tipo de arquitecturas tienen mucho más potencial que para ejecutar solo este tipo de procedimientos. Por otro lado, aunque en costes salga rentable porque se paga por uso, si sólo se necesita ejecutar procedimientos almacenados hay aplicaciones que proporcionan servidores de automatización muy exitosos como puede ser el caso de *Jenkins o Rundeck* , este último además es *open source.*

## 6.6 Ventajas

Hasta ahora se ha tratado de definir las arquitecturas *serverless* y describir sus características para que se pueda entender cómo funciona este modelo y lo que se debe tener en cuenta para llevarlo a la práctica, sin embargo, no se recomienda a nadie decidir el uso del modelo *serverless* sin hacer un balance de pros y contras.

### 6.6.1 Reducción de costes

La reducción de costes es una de las grandes ventajas que nos ofrecen este tipo de arquitecturas, así lo plasman Gojko Adzic y Robert Chatley en *Serverless computing: Economic and Architectural Impact* [21] en un estudio sobre los principales servicios de *hosting.*

#### 6.6.1.1 Reducción de costes operacional

*Serverless* es en su forma más simple es una solución de *outsourcing*, es decir, te permite subcontratar el servicio pagando a un proveedor para manejar bases de datos, servidores y la lógica de aplicación que en otro caso tendrías que administrar tú. En esencia, la reducción del coste operacional se da gracias al efecto que se denomina *economy of scale effect,* el cual se da por el uso de un servicio predefinido que muchos otros usan, por ejemplo, pagar menos por el uso de una base de datos porque el proveedor tiene corriendo miles de bases de datos similares a la vez.

La reducción de costes en este aspecto se puede diferenciar en dos puntos diferentes. El primero, la reducción de costes por compartir la infraestructura con otras personas al usar un proveedor de servicio, como en el caso de *hardware* y *networking* tal como se ha descrito con el concepto de *economy of scale effect.* El segundo, son los costes laborales que se ahorra de construir una arquitectura *serverless* para a una aplicación frente a una arquitectura tradicional donde tendríamos que pagar a un conjunto de personas para que puedan construir, administrar y mantener toda la infraestructura.
Cabe añadir, que con una arquitectura *serverless* no eliminamos toda la administración, es muy probable que debamos tener un equipo de operaciones y arquitectura que lo administre, pero el coste va a ser mucho menor ya que como decía anteriormente nos ahorramos tener que comprar servidores, pagar su instalación, el mantenimiento de los servidores, pagar la construcción y administración de entornos de trabajo y de ejecución desde cero.

Este tipo de beneficio, en cambio no es tan relevante cuando se compara con servicios *IaaS o servicios PaaS,* porque ya se benefician de la reducción de costes por compartir la infraestructura y del ahorro de costes laborales, ya se sigue teniendo la infraestructura en el proveedor, la diferencia de costes en este caso se reduciría a detalles menores como la cantidad de administradores que se necesiten para poder administrar la infraestructura *cloud* y todos los aplicativos, y si es más barato mantener máquinas corriendo todo el día o funciones que se ejecuten bajo demanda.

#### 6.6.1.2 Reducción de costes de desarrollo

Por otro lado, y relacionado con lo anterior, las arquitecturas *serverless* a diferencia de las *PaaS* y las *IaaS* tratan como una mercancía a todos los componentes de una aplicación, mientras que estas dos últimas sólo mercantilizan la administración del servidor y el sistema operativo y supone un gran ahorro a la hora de referirse a los costes de desarrollo.

¿Qué quiero decir con mercantilizar y qué relación tiene con los costes? Lo mismo que me refería con el *economy of scale effect*, comprar un producto que un proveedor vende de forma masiva.
Las arquitecturas *serverless* se benefician de este efecto al máximo, ya que como explique al principio del trabajo se componen de servicios *FaaS* y *BaaS,* esto permite que estas arquitecturas no sólo se beneficien del ahorro que supone contratar por uso un servidor a un proveedor, sino que además pueden contratar más servicios, esto supone un ahorro ya que al hacer uso de este servicio no es necesario su desarrollo y por lo tanto supone un ahorro de cara a los recursos.
Un ejemplo de este tipo de ahorros en el desarrollo es un sistema de autenticación. Muchas aplicaciones usan sistemas de autenticación y para ello desarrollan el suyo propio, el cual suele constar de registro de usuarios, *login*, *logout*, administración de contraseñas, integración con otros proveedores, autenticación, etc. Pero por el contrario, existen servicios como *Auth0* que ofrecen una funcionalidad muy completa ya que se dedican sólo a vender y mantener un servicio de autenticación, por lo que están más especializados de una forma muchos más barata y cómoda que el desarrollo de este tipo de módulos. Cabe recalcar que *Atuh0* se puede integrar en las principales plataformas *cloud* y en gran cantidad de aplicaciones conocidas.
Otro ejemplo son las bases de datos *BaaS* como por ejemplo *Firebase* las cuales tienen integración con aplicaciones *serverless* y el cual permite administrar una base de datos eliminando gran parte de la complejidad de manejar una convencional.

En cualquier caso, aunque este tipo de soluciones parece ser buena para ahorrar costes dependiendo del contexto en el que se use y, en general como esté construida la arquitectura puede aumentar la complejidad y retorcer mucho las soluciones empleadas, aunque esto lo trataré en la sección de desventajas.

#### 6.6.1.3 Reducción de costes de escalado

Uno de los puntos donde más costes puede ahorrar usar una arquitectura *serverless* frente a una tradicional es en el escalado, como ya he explicado en el trabajo uno de los puntos fuertes de estas arquitecturas es su escalado, este se realiza de forma automática por el proveedor, es totalmente flexible y con una granularidad muy fina, llegando a escalar en función de las consultas que se hagan al sistema. Si a lo anterior se le suma el factor de pagar solo por lo consumido el ahorro de costes puede llegar a ser considerable.

A continuación, se van a exponer dos ejemplos donde se discute mejor esta ventaja.

Ejemplo 1: en el supuesto de que se tiene una aplicación que procesa cada consulta en 50ms y de media el uso de la unidad central de procesamiento (en adelante CPU) por hora es de un 1%.

En este caso si se tuviese una máquina dedicada para alojar esta aplicación, como en los modelos tradicionales, sería altamente ineficiente, ya que se paga todo el uso que se hace de la máquina a pesar de que la mayoría del tiempo está ociosa.
Una posibilidad en este caso sería alojar la aplicación en una instancia con menos recursos para ajustarlo al uso que se da de la misma, y de esta forma reducir el coste de una máquina tan sobre ajustada. De todas formas, el tiempo que la máquina esté ociosa va a seguir generando gastos, por eso mismo en este caso las tecnologías *FaaS* son una buena solución, ya que se paga solo por el uso de la aplicación, siguiendo con el ejemplo anterior se pagarían 100ms por minuto, lo que equivale a un 0,15% del tiempo.

El anterior ejemplo, aunque está llevado al extremo es extrapolable a todas las aplicaciones que tengan tiempo ocioso. Y si se valora una migración a un sistema *serverless* por este motivo, se ha de calcular minuciosamente el coste por unidad de tiempo que el sistema está ocioso en el caso de las arquitecturas tradicionales y compararlo por el precio por unidad de tiempo en el servicio *FaaS,* cabe destacar que este último coste es complejo de calcular ya que se paga por cantidad de memoria usada para correr la función, tiempo que tarda en ejecutarse y número de solicitudes.

Ejemplo 2: en el supuesto que se tiene una aplicación que da servicio a un alto número de usuarios, como por ejemplo una de banca. El banco tiene en su aplicación una media de 300 consultas por minuto, pero el primer día del mes que se suelen ingresar las nóminas, rentas y pensiones, esto produce que el sistema, en vez de tener 300 consultas por minuto tenga 3000, lo que implica que la carga en el sistema es 10 veces mayor de lo habitual. Este hecho puede producir en un sistema tradicional tiempos de respuesta muy altos e incluso la pérdida del servicio de la aplicación.
Para solventar este problema en un sistema tradicional tendríamos que soportar un factor 10 veces mayor de carga que lo habitual con los costes que esto conlleva, en cambio, con un sistema *serverless* este hecho no supondría un problema, ya que en los puntos que se dispare el uso de la aplicación se pagaría por la computación extra y escalaría automáticamente.

Un sistema más moderno como puede ser sistemas de contenedores o contenedores *cloud* también pueden ser una buena solución si estos sistemas cuentan con auto escalado, aunque escalarían de forma más lenta y con una menor granularidad frente a un sistema *serverless* solamente por el hecho de tener que arrancar y crear los contenedores necesarios para hacer frente a la carga.

Normalmente, a no ser que se tenga un tráfico consistente y la aplicación aproveche al máximo los recursos del servidor, un *host serverless* va a suponer un ahorro en este apartado. Aun así, salvo casos obvios será necesario calcular los costes de los sistemas que se valora utilizar.

#### 6.6.1.4 Optimización como medida de reducción de costes

Las arquitecturas *serverless* y en esencia los servicios *FaaS* introducen una nueva manera de reducir costes a través de la optimización de código, y aunque esta optimización se puede llevar a cabo en sistemas más tradicionales el ahorro no es tan claro como con estos servicios.

En estas arquitecturas optimizar el código para que tarde menos en ejecutarse es una forma de ahorrarse costes, ya que las funciones lambda sobretodo se pagan por la duración de la ejecución además de la memoria usada.
Por ejemplo, si tenemos una función que tarda en ejecutar 1 segundo y optimizando el código se reduce el tiempo a 200ms, eso ya significa que se están reduciendo un 80% los costes sin cambios de infraestructura y si además se optimiza para hacer uso de la memoria justa se puede reducir aún más los costes.

### 6.6.2 Facilidad de gestión operacional

Al principio del trabajo se expone que las arquitecturas *serverless* se componen de dos servicios *cloud*, *BaaS* y *FaaS.*
En el caso de la parte *BaaS* es obvio que se requiere menos gestión operacional ya que al admitir menos componentes se necesita menos trabajo.
Sin embargo, en la parte *FaaS* hay algunos aspectos que merecen la pena ser tratados de forma independiente.

#### 6.6.2.1 Beneficios operacionales de escalado

Este punto se va a basar en algunas premisas del apartado de reducción de costes de escalado, ya que, al ser automático, el escalado también genera nuevas ventajas en la operación de los sistemas reduciendo su administración.

Un proceso de escalado necesita que un operador elimine o añada una o varias instancias a un conjunto de servidores. Con *FaaS* esta administración se elimina en su totalidad, ya que escala solo a nivel de consulta.

Por otro lado, existen arquitecturas que también cuentan con auto escalado, pero no eliminan completamente la administración que se necesita sobre esos sistemas, ya que requieren una preparación y un mantenimiento. Por ejemplo, es el caso de los servicios de contenedores en la nube como *ECS,* el cual requiere una configuración básica, que indique el número de tareas deseadas corriendo en el *clúster*, en qué tipo de instancias van a correr, tipo de servicio, etc.

#### 6.6.2.2 Reducción de la complejidad de empaquetado y despliegue

Empaquetar y desplegar una función *FaaS* es mucho más simple que desplegar un servidor entero, o que desplegar un aplicativo en contenedores.
En *FaaS* se puede desplegar un aplicativo en cuestión de segundos, ya que se puede desplegar simplemente a partir de un archivo de tipo .zip, de un repositorio de código o de una imagen creada previamente. Esto es mucho más rápido que desplegar un aplicativo

en un servidor monolítico o en celdas de un servidor, que pueden requerir horas. Incluso es más rápido que desplegar un aplicativo en un sistema de contenedores que puede tardar entre 5 y 10 minutos si se usa un *pipeline* con herramientas de automatización que genere la imagen a desplegar y scripts que paren las versiones antiguas de los contenedores y arranquen contenedores con la nueva versión del aplicativo.

Además, normalmente en los servicios *FaaS,* como *AWS Lambda* permiten editar el propio código en la consola proporcionada por el proveedor, aunque esto no es recomendable para entornos productivos.

#### 6.6.2.3 Experimentación continua y *time to market*

Una ventaja más orientada al rendimiento en empresas tecnológicas es la mejora de estos sistemas al *time to market*, esto es el tiempo que transcurre desde que se concibe un producto hasta que está disponible para la venta.
Los servicios *FaaS* agilizan mucho este proceso ya que permiten que se pueda probar continuamente nuevas ideas y rápidamente actualizar los sistemas en producción y preproducción, además se acopla perfectamente a metodologías ágiles de trabajo.

Aunque el concepto de *continuous delivery* ya permite iterar sobre proyectos estables de forma rápida, estas tecnologías tienen una gran capacidad para probar nuevas ideas por un coste mínimo y gran comodidad, esto dota a las arquitecturas *serverless* de mucha versatilidad para una continua experimentación.

### 6.6.3 Computación más ecológica

Uno de los problemas de la sociedad actual es el impacto al medio ambiente que produce el ser humano, como el cambio climático y la contaminación, los cuales están directamente relacionados.

Según la revista *Forbes* en 2015 afirma "*Typical servers in business and enterprise data centers deliver between 5 and 15 percent of their maximum computing output on average over the course of the year.*" [22]

La infraestructura en la nube frente infraestructuras tradicionales aprovecha mejor los recursos maximizando la utilidad y el uso, esto reduce el impacto en el medio ambiente, además fuerza a que las empresas compren servidores solo cuando de verdad lo necesitan.
Normalmente, cuando se decide que solución se le da a la infraestructura de una aplicación no se piensa en el impacto medioambiental de la electricidad que se necesita y los combustibles fósiles que se usan para dar electricidad a los *datacenters.* Con soluciones como las arquitecturas *serverless* se depositan estas decisiones en el proveedor de servicio que ajusta la potencia al uso que se necesita y se evita caer en sobre provisionamiento.

## 6.7 Inconvenientes

Es posible que durante la lectura del trabajo haya parecido que esta tecnología es perfecta, a pesar de que se ha intentado resaltar que no siempre este tipo de modelo va a ser la solución a todos los problemas.

Por si no han quedado claras estas carencias, en este apartado se recogerán todos los inconvenientes de este modelo, algunos de ellos inherentes al mismo y por lo tanto no pueden ser corregidos a futuro, ya que están ligados a la idea de este modelo, y los restantes son dependientes de la implementación y pueden ser resueltos a futuro.

### 6.7.1 Inconvenientes inherentes al modelo

#### 6.7.1.1 Control del proveedor

El aspecto negativo de usar tecnologías alojadas en la nube es que se está sujeto al control del proveedor de servicio, esta falta de control muchas veces se manifiesta en límites inesperados en el servicio, cambios en los costes, pérdida de funcionalidad, actualizaciones forzadas, etc.

Como relataba *Charity Majors* en su artículo sobre las mejores prácticas operacionales en sistemas *serverless* "The service, if it is smart, will put strong constraints on how you are able to use it, so they are more likely to deliver on their reliability goals. **When users have flexibility and options it creates chaos and unreliability.** If the platform has to choose between your happiness vs thousands of other customers' happiness, they will choose the many over the one every time — as they should." [23]

Los proveedores de servicios generalmente imponen grandes restricciones, ya que cuando se otorga flexibilidad al usuario, este genera caos y falta de fiabilidad.

#### 6.7.1.2 Inconvenientes de tecnologías de uso compartido

Las tecnologías de uso compartido se refieren a sistemas que son compartidos por otros usuarios, normalmente se denomina a este concepto como *multitenancy*. Por ejemplo, cuando dos aplicaciones diferentes de distintos usuarios corren en la misma máquina.
Esta práctica es común en servicios en la nube para optimizar el uso de los sistemas, por otro lado, el proveedor se encarga de que en la medida de lo posible de que el usuario no se de cuenta de que comparte recursos con otro cliente.

En el caso de *FaaS* se trata de un servicio de uso compartido, es decir, dos funciones de diferentes clientes se pueden llegar a ejecutar en la misma instancia, esto puede acarrear ciertos problemas. El primero es de seguridad, en alguna ocasión se puede dar el caso que un cliente es capaz de ver el software de otro cliente con el que comparte instancia. El segundo son los problemas de rendimiento, ya que es posible que durante una carga de trabajo de una función que comparte instancia con otra función que ya se está ejecutando, se produzca una ralentización. Y, por último, se pueden producir problemas en la robustez del sistema, produciéndose un error en una función de un cliente que produce un error en la función de otro cliente.

Estos errores, en general, se pueden dar en todas las tecnologías de uso compartido, por el contrario, los principales proveedores tienen servicios *serverless* muy robustos y avanzados que no suelen tener este tipo de fallos. De hecho, en mi experiencia durante este trabajo y un año trabajando con *AWS Lambda* no he visto que se haya producido ninguno de estos errores.

#### 6.7.1.3 Dificultad para migrar y dependencia del proveedor

Un gran problema de estas arquitecturas es que están ligadas a un proveedor de servicio y las características de los servicios *FaaS* en diferentes proveedores tienen muchas diferencias. Esto tiene un gran impacto si se decide migrar de proveedor, lo que genera una dependencia al mismo.
Para realizar una migración de un proveedor a otro en primer lugar será necesario cambiar las herramientas operacionales que se encargan del despliegue de los aplicativos, monitorización, almacenamiento de *logs*, etc. A no ser que estas sean externas al proveedor y se puedan integrar en el nuevo sistema. En segundo lugar, si no acepta el lenguaje en el que está la función será necesario migrar el código para utilizar un lenguaje que sea aceptado por el nuevo proveedor, en cambio si el leguaje fuese aceptado por ambos proveedores sería necesario modificar las funciones para que puedan correr con la nueva interfaz *FaaS* y la nueva *API* del proveedor. En tercer lugar, es probable que haya que cambiar el diseño y la arquitectura si existiesen diferencias en el funcionamiento de los componentes. Si se usase una arquitectura *serverless* medianamente compleja hay que migrar todos los elementos al nuevo proveedor, no solo la componente *FaaS*.

Para contrarrestar este problema, personas y empresas que adoptan un enfoque *multi-cloud*, de forma que el código es genérico y las herramientas operacionales son agnósticas al proveedor con el que se trabaja. Este enfoque, aunque permite el migrado del sistema de forma más sencilla es mucho más costoso debido a que la mayor parte de herramientas operacionales van a tener que ser externas y es necesario condicionar toda la programación de forma que se vuelve más compleja.

El planteamiento inicial antes de desarrollar una aplicación en un sistema *serverless* para evitar una migración es estudiar exhaustivamente todas las necesidades de la aplicación y los elementos que proporciona el proveedor para no reducir la posibilidad de migrado. Una tendencia que suelen realizar las empresas tecnológicas es elegir el proveedor con el que están más contentos, adaptar sus sistemas a los productos que ofrece y explotar al máximo todas sus capacidades.
En cualquier caso, si se desea profundizar en este tema y construir un sistema *serverless* lo más migrable posible recomiendo encarecidamente la lectura de “Mitigating serverless lock-in fears” de Wisen Tanasa [24], la cual también me ha servido para desarrollar este apartado.

#### 6.7.1.4 Aumento del riesgo en la seguridad informática

Con el nacimiento y la adopción de nuevos modelos informáticos, como es el caso de las arquitecturas *serverless*, nacen nuevas preocupaciones en el área de la seguridad informática.

- En el caso de que cualquier arquitectura *cloud* como puede ser *serverless* utilice varios proveedores, provoca que tengan que haber varias implementaciones de seguridad. Este hecho aumenta la superficie en la que se pueden realizar ataques y, por lo tanto, aumenta la probabilidad de que haya un ataque satisfactorio.
- En el caso de necesitar bases de datos de tipo *BaaS* en la arquitectura, provoca que se pierda la barrera de protección que hay en el lado del servidor en arquitecturas tradicionales.
- Cuando se adoptan este tipo de arquitecturas en empresas, comienzan a surgir un desarrollo masivo de funciones. Estas funciones van ligadas, en el caso de *AWS* a políticas de administración de identidades (a partir de ahora *IAM*) y roles *IAM* que limitan los permisos, los usuarios y las acciones pueden llevar a cabo los usuarios. El caso es que en este tipo de sistemas se pueden cometer fallos con mucha facilidad y provocar una falla de seguridad, por lo que hay que ser cuidadoso y tener todas las consideraciones posibles.

#### 6.7.1.5 Pérdida de optimización en el lado del servidor

Cuando se utilizan arquitecturas *serverless* que se componen de servicios *FaaS y BaaS* no es posible optimizar la parte del servidor, ya que esta pertenece al proveedor y no está al alcance de los usuarios. Por ello, no es posible adoptar los patrones llamados *back end for front end*, los cuales optimizan la aplicación en el lado del servidor y provoca que las operaciones en este se ejecuten más rápido y en caso de aplicaciones móviles consuman menos batería.

Una posible mitigación para este tipo de problemas es implementar funciones *FaaS* más ligeras o patrones de lógica del servidor más ligeros.

#### 6.7.1.6 No se conserva el estado

En la descripción de características de las arquitecturas *serverless* en el documento ya se había hablado de que estas arquitecturas no guardan el estado y es mejor trabajar asumiendo este hecho.

A pesar de esto, ya se trató el hecho de que en el caso que fuese necesario guardar el estado se pueden utilizar bases de datos *SQL*, *NoSQL*, servicios externos de cachés o almacenamiento de objetos para contrarrestar este inconveniente, aunque esto siempre produce un retraso por ser servicios de almacenamiento externo a nuestra aplicación. Por otro lado, *FaaS* permiten el uso de la cache local siempre y cuando las funciones sean usadas muy frecuentemente, sin embargo, también se pueden usar caches de latencia baja como *Redis* o *Memcache*, esto puede suponer trabajo extra, aumento de los costes y dependiendo de los casos de uso pueden ser muy lentas.

### 6.7.2 Inconvenientes de implementación

Estos tipo de inconvenientes surgen dependiendo del estado del arte y los proveedores y la comunidad intenta buscar soluciones.

#### 6.7.2.1 Denegación de servicio

En general todos los servicios *FaaS* tienen un límite de concurrencia, es decir, un número de instancias de la misma función corriendo de forma simultánea. En el caso de *AWS* se pueden ejecutar hasta mil funciones de forma concurrente por cuenta, este hecho puede derivar en una denegación de servicio si no se controla adecuadamente este límite. Muchas empresas u organizaciones comparten la misma cuenta de producción que de pruebas y este hecho puede hacer que se alcancen las mil funciones ejecutándose de forma concurrente de forma que para las nuevas peticiones se realice una denegación de servicio.
Para evitar este problema se recomienda usar diferentes cuentas para cada proyecto distinguiendo además entre entornos productivos y entornos pre productivos.

La solución administrativa para controlar el número de ejecuciones concurrentes por función es usar, en el caso de *AWS Lambda,* el parámetro *Reserve concurrency*, el cual establece un número máximo de ejecuciones de una función de forma concurrente.

En el peor de los casos se le puede solicitar al soporte del proveedor que te aumenten el límite de ejecuciones simultáneas de funciones de mil a un nuevo valor mayor, pero esta no es una buena práctica y usualmente solo reduce un poco la probabilidad de que se produzca una denegación de servicio.

#### 6.7.2.2 Duración de la ejecución limitada

Como máximo las funciones lambda tienen quince minutos para ejecutarse, este hecho produce que numerosas aplicaciones de larga duración no puedan implementarse en este tipo de arquitecturas. Si quedase alguna duda respecto a este apartado sugiero ir al apartado 5.1.2 donde se explica mejor esta característica.

#### 6.7.2.3 Latencia de arranque

Este inconveniente se refiere a lo relatado en el apartado 5.1.3, en el cual se explica que los servicios *FaaS*, al ejecutarse sobre contendores y ser administrado de forma automática por el proveedor pueden sufrir una latencia de arranque o arranques en frío.

Los arranques en frío se producen cuando una función que se ejecuta sobre un contenedor que se invoca después de un periodo de tiempo lo suficientemente alto respecto a la anterior invocación, lo que produce que el proveedor retire ese contenedor que no se estaba usando, por lo que el proveedor ante esta nueva invocación tiene que volver a arrancar un nuevo contenedor y inicializarlo, lo que produce un retraso en la ejecución del sistema y pudiendo afectar al tiempo de respuesta de la aplicación.

Como *workaround* se puede hacer uso de *keep alives*, es decir, llamadas falsas a la función para que el proveedor nunca retire el contenedor y, por lo tanto, no producir arranques en frío.
Si se desea profundizar en esta desventaja sugiero ir al apartado 5.1.3 donde se encuentra ampliamente explicado.

#### *6.7.2.4* Testing en arquitecturas *serverless*

Las pruebas unitarias en estos sistemas no suponen un problema, ya que como en una aplicación normal consiste en poner a prueba el funcionamiento de las funciones del aplicativo que se quiere probar. En cambio, las pruebas de integración en estos sistemas son mucho más complejas, ya que dependiendo del proveedor pueden surgir dudas como por ejemplo si la integración debe usar sistemas externos o si será fácil integrar la aplicación o si costará mucho parar y arrancar el sistema o si incluso hay alguna tarifa especial para pruebas.
Algunos proveedores como AWS o Azure permiten ejecutar localmente la función a probar, pero esto no es recomendable ya que difiere mucho un entorno local de uno en la nube, estas pruebas se deberían hacer en entornos previos y a ser posible que formen parte del *pipeline* de despliegue de aplicativos, de esta forma se depurarán los errores más rápidamente.

#### *6.7.2.5* Carencias en la depuración de errores

La depuración de errores en servicios *FaaS* es uno de los grandes problemas de estos sistemas, ya que debido a la falta de acceso a nivel de sistema operativo es casi imposible depurar con herramientas de depuración.

*AWS* y *Azure* permiten bajar la aplicación a nuestro entorno local y ahí hacer la depuración, *Azure* de hecho tiene mejor implementado este sistema de depuración ya que permite disparar eventos en producción y de esta forma probar de una forma más a fin al funcionamiento en un entorno *cloud.*

El resto de las formas de depuración se consideran de tipo *workaround*, es decir, acciones que sirven para depurar, pero no son formas de depurar. Por ejemplo, AWS permite disparar eventos de prueba mientras se desarrolla la función para comprobar errores en tiempo de desarrollo. Los *logs* en el caso de *AWS* se almacenan en el servicio *Cloudwatch logs*, que permite comprobar los errores que se han producido durante la ejecución de la aplicación. El servicio *AWS x-ray* permite diagnosticar errores y problemas de rendimiento producido por una aplicación o un servicio como puede ser *AWS Lambda.*
Otras herramientas que pueden ser de gran ayuda para depurar son los *Application Performance Managemente o APM*, los cuales son herramientas muy avanzadas de monitorización de aplicaciones, gracias a los agentes de estos programas capturan una gran cantidad de datos sobre las aplicaciones y en ellos podemos ver los errores que hay en el tiempo, las trazas de error completas, en que punto de la transacción se ha producido el error, etc. Esto nos permite depurar errores, pero por el contrario este tipo de aplicaciones tienen un coste muy alto.
Existen aplicaciones orientadas sólo a depurar funciones *serverless* que son opciones más sencillas y baratas comparado con un *APM*, como por ejemplo *Rookout, Epsagon* o *Lumigo*, estas aplicaciones permiten hacer *troubleshooting* de errores, prever errores antes de que sucedan y recolectan datos para facilitar el desarrollo y pruebas en este tipo de aplicaciones.

#### 6.7.2.6 Monitorización

Por la naturaleza de los servicios *FaaS* y los contenedores en las que estas corren es normal que la monitorización sea compleja, además al no tener acceso al sistema operativo de la instancia hace muy complicada la monitorización los servicios *FaaS*.

Los proveedores como *AWS* proporcionan una monitorización básica de las funciones con datos como el número de invocaciones, la duración de cada invocación, el número de errores, tasa de éxito, etc. Por lo que si queremos una monitorización más detallada sería necesario usar un *APM* y esto incrementaría considerablemente los costes, además la monitorización no es tan buena como con una arquitectura basada en microservicios o un monolito ya no se tiene acceso al sistema operativo y ni a las *APIs*.

#### 6.7.2.7 Subestimar u olvidar la operativa

En muchos artículos, incluso al principio del trabajo se puede percibir que el modelo *serverless* no necesita la administración por parte de los departamentos de operaciones, en cambio, todavía se necesitan para la monitorización, el escalamiento de la infraestructura, la seguridad y el *networking* de la plataforma.
Con este tipo de modelos puede parecer que el trabajo de operaciones es prescindible, pero esto puede llevar a un falso sentimiento de seguridad cuando todavía existen vulnerabilidades y errores, la mejor forma de que se produzcan problemas es que los equipos de desarrollo tengan en consideración al equipo de operaciones y permitir que administren toda la infraestructura.

# 7 Pruebas y resultados

Estas pruebas son pruebas de concepto que se han realizado para contrastar datos leídos en la bibliografía o para probar las características principales, se han realizado diversas pruebas de concepto para validar los distintos aspectos que se discuten en el TFG, estas pruebas se presentan en el anexo A.

# 8 Conclusiones y trabajo futuro

## 8.1 Conclusiones

Las arquitecturas *serverless* no son la solución definitiva. Hay que considerar estos sistemas como una posible solución más a nuestros problemas, lo que da más diversidad a nuestras posibles soluciones y por lo tanto puede tener como resultado soluciones más adecuadas.

Hay que tener en cuenta que, aunque se trabaje con un tipo de arquitecturas relativamente novedoso las ventajas de emplear este modelo no van a superar siempre a los inconvenientes. Entendiendo como funcionan las arquitecturas *serverless* y sus propiedades, debemos hacer un balance de ventajas e inconvenientes y comprobar que no haya ningún requisito bloqueante, es decir, un requisito del problema que es necesario que se satisfaga y, por lo tanto, que descarte estos sistemas como una posible solución.

Por ejemplo, si necesitamos implementar un sistema de procesamiento de datos sensibles, que esté constantemente procesando datos y que permita ser migrado de un *cloud* a otra *cloud*, es probable que las arquitecturas *serverless* no sean la solución adecuada, ya que se necesita un grado alto de seguridad, ejecución constante del aplicativo y es susceptible de ser migrado.

Por otro lado, si no hay ningún requisito bloqueante, se debe hacer un balance de ventajas e inconvenientes en función de las características del proyecto. Por ejemplo, si un banco quiere con la mayor celeridad posible y costes mímos sacar un producto de cálculo de hipotecas dentro de su web una de las soluciones posibles son las arquitecturas *serverless,* dado que va a reducir el *time to market* y fomentar metodologías ágiles de trabajo, aguantará cargas desiguales de trabajo en función de la gente que lo está usando y reducirá los costes.

Realizando un balance general, el modelo *serverless* es un modelo muy eficiente en la reducción de costes, ya que usa la filosofía *cloud* de solo pagar por el uso que se haga, facilitando la operativa con un muy eficiente escalado automático, reduciendo la complejidad del pipeline de *CICD y* reduciendo el *time to market*. Por el contrario, se pierde parte del control de la aplicación, se aumenta el riesgo de la seguridad informática, se pierde rendimiento en el lado del servidor, condiciona la programación al no conservar el estado, puede tener arranques lentos produciendo latencia, posee un tiempo limitado de ejecución y es muy ineficiente para la realización de pruebas, depuración y monitorización. Esto se muestra de forma ordenada en la siguiente tabla.
A pesar de todos los aspectos negativos, este modelo sigue creciendo y al que se están adhiriendo multitud de sistemas de información por que la mayoría de los aspectos negativos se pueden arreglar a cambio de aumentar la complejidad de la arquitectura y los que no son solucionables no son tan importantes como los aspectos positivos que son muy atractivos sobretodo para los departamentos de negocio, ya que aseguran un *time to market* mínimo y reducción de costes. Aun así, es un modelo que sigue desarrollándose para cubrir cada vez más casos de uso y hoy en día es una solución que se tiene en cuenta cuando se trabaja con sistemas en la nube por lo que su crecimiento depende del desarrollo del producto y del crecimiento de las tecnologías en la nube.

| VENTAJAS | INCONVENIENTES |
|---|---|
| **Reducción de costes:**<br>- Reducción de costes operacional.<br>- Reducción de costes de escalado.<br>- Reducción de costes de desarrollo.<br>- Reducción de costes por optimización. | **Inherentes al modelo:**<br>- Control del proveedor.<br>- Problemas de uso compartido.<br>- Dificultad de migrado y dependencia del proveedor.<br>- Aumento de riesgo de la seguridad informática.<br>- Perdida de optimización.<br>- No conserva el estado, condiciona la programación. |
| **Facilidad de gestión operacional:**<br>- Escalado automático y eficiente.<br>- Reducción de la complejidad de empaquetado y despliegue.<br>- Mejora en la experimentación continua y time to market. | **De implementación:**<br>- Denegación de servicio.<br>- Duración limitada de la ejecución.<br>- Latencia de arranque.<br>- Dificultad de testing.<br>- Dificultad de debugging.<br>- Monitorización pobre. |
| **Computación más ecológica.** | - |

Tabla 3. Comparación entre las principales ventajas y desventajas de las arquitecturas *serverless.*

En cualquier caso, hay que tener en cuenta que las arquitecturas *serverless* siguen mejorando, ya que las tecnologías *FaaS* aun están en una fase inmadura y se esperan mejoras, por lo que puede que estos sistemas sean una solución habitual en el futuro, hasta entonces no son la solución definitiva, solo una más.

## 8.2 Trabajo futuro

En primer lugar, como trabajo a futuro se propone ampliar el trabajo estudiando y probando más a fondo las arquitecturas *serverless* en la nube privada y híbrida, poniendo también a prueba los *frameworks* más utilizados en este tipo de nubes, estudiar sus características y como condicionan a las características generales del modelo.
Además, en relación con lo propuesto anteriormente podría ser interesante una comparación entre estos tres tipos de modelos, nube pública, nube privada y nube híbrida, para saber dónde destacan cada uno de ellos y poder generar una conclusión sobre que tipo de nube se debe elegir según qué proyectos.

En segundo lugar, se propone ampliar el trabajo a más tipos de servicios *FaaS,* sin centrarnos sólo en *AWS Lambda* y en los principales proveedores, para dotar al trabajo de una visión más general de estas tecnologías y valorar de forma más amplia todas sus capacidades y limitaciones, y, de esta forma comprobar en que destaca cada uno.

En tercer lugar, se propone añadir al trabajo un estudio de la base de datos *AWS Aurora serverless*, que es un producto creado para fomentar la idea de *serverless* en bases de datos, pero que no tiene porque estar orientado sólo para estas arquitecturas. Aun así, me parece interesante incluirlo en el estudio debido a la afinidad con estas arquitecturas y la mejora que puede suponer este tipo de infraestructura en las arquitecturas *serverless*.

En cuarto, se propone como ampliación del trabajo incluir el servicio *AWS Fargate,* que une la idea de modelo *serverless* a servicios *cloud* de contenedores, dando lugar a sistemas de microservicios *serverless*. Además, se pueden comparar estos sistemas de microservicios *serverless* con los sistemas de microservicios *serverless* montados a partir de servicios *FaaS,* dando lugar a una discusión sobre dos tipos de soluciones *serverless* diferentes.

En quinto y último lugar, este TFG me ha proporcionado una base respecto a arquitecturas *serverless* y tecnologías *cloud* que me pueden ser útiles para futuros proyectos y proporcionándome nuevas oportunidades.

**Referencias**


[1] M. Roberts, «Serverless Architectures,» 22 May 2018. [En línea]. Available: https://martinfowler.com/articles/serverless.html.

[2] M. Roberts, «Learning Lambda,» 9 November 2017. [En línea]. Available: https://blog.symphonia.io/posts/2017-11-09_learning-lambda.

[3] M. Roberts y J. Chapin, What is Serverless?, Sebastopol: O'REILLY, 2017.

[4] C. Gallego, F. Álvarez y M. Evgeniev, «Serverless,» 2017. [En línea]. Available: https://www.bbva.com/es/serverless/.

[5] Á. Alda Rodríguez, F. Álvarez, G. Díaz López de la Llave, M. Evgeniev y P. Horrillo, «La Economía de las Arquitecturas 'Serverless',» 2018 june 25. [En línea]. Available: https://www.bbva.com/es/economia-arquitecturas-serverless/.

[6] T. Lynn, P. Rosati, A. Lejeune y V. Emeakaroha, «A preliminary Review of Enterprise Serverless Cloud Compunting (Function-as-a-Service) Platforms,» *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom),* pp. 162-169, 2017.

[7] G. M. a. P. R. Brenner, «Serverless Computing: Design, Implementation, and Performance,» *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW),* pp. 405-410, 2017.

[8] P. Sbarski, Y. Cui y A. Nair, Serverless Architectures on AWS, Manning publications, 2020.

[9] W. Fellows, «A sea of change – Migrating Workloads and applications to the cloud,» 12 Septiembre 2018. [En línea]. Available: https://www.brighttalk.com/webcast/10363/318591?utm_source=451social.

[10] L. Columbus, «State of Enterprise Cloud Computing,» 30 August 2018. [En línea]. Available: https://www.forbes.com/sites/louiscolumbus/2018/08/30/state-of-enterprise-cloud-computing-2018/#609dc9d0265e.

[11] K. Costello, «Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019,» 2 April 2019. [En línea]. Available: https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g.

[12] L. Columbus, «83% Of Enterprise Workloads Will Be In The Cloud By 2020,» 7 January 2018. [En línea]. Available: https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/#4b74902e6261.

[13] S. Moore, «Gartner Says 28 Percent of Spending in Key IT Segments Will Shift to the Cloud by 2022,» 18 September 2018. [En línea]. Available: https://www.gartner.com/en/newsroom/press-releases/2018-09-18-gartner-says-28-percent-of-spending-in-key-IT-segments-will-shift-to-the-cloud-by-2022.

[14] Stackoverflow, «Most Loved, Dreaded, and Wanted Platforms,» 2017. [En línea]. Available: https://insights.stackoverflow.com/survey/2017#technology-_-most-loved-dreaded-and-wanted-platforms.

[15] Stackoverflow, «Most Loved, Dreaded, and Wanted Platforms,» 2018. [En línea]. Available: https://insights.stackoverflow.com/survey/2018#technology-_-most-loved-dreaded-and-wanted-platforms.

[16] P. Mell y G. Timothy, «The NIST Definition of Cloud Computing,» National Institute of Standarts and Technology, 2011.

[17] Bikeborg, Artist, *Cloud Compunting layers.* [Art].

[18] O. Wolf, «Serverless Architecture in short,» 2020. [En línea]. Available: https://specify.io/concepts/serverless-baas-faas.

[19] Gartner, «magic quadrant for cloud infrastructure as a service worldwide 2019,» 2019. [En línea]. Available: https://cloud.google.com/gartner-cloud-infrastructure-as-a-service?hl=es-419.

[20] c. Adrian, 28 May 2016. [En línea]. Available: https://twitter.com/adrianco/status/736553530689998848.

[21] G. Adzic y R. Chatley, «Serverless Computing: Economic and Architectural Impact,» *ESEC/FSE 2017: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering,* pp. 884-889, 2017.

[22] B. Kepes, «30% Of Servers Are Sitting "Comatose" According To Research,» 2015 june 3. [En línea]. Available: forbes.com/sites/benkepes/2015/06/03/30-of-servers-are-sitting-comatose-according-to-research/#1286e2ad59c7.

[23] C. Majors, «Operational Best Practices #Serverless,» 2016 May 31. [En línea]. Available: https://charity.wtf/2016/05/31/operational-best-practices-serverless/.

[24] W. Tanasa, «Mitigating serverless lock-in fears,» 21 march 2019. [En línea]. Available: https://www.thoughtworks.com/es/insights/blog/mitigating-serverless-lock-fears.

[25] C. Boulton, «Serverless: The future of cloud computing,» 13 February 2019. [En línea]. Available: https://www.cio.com/article/3244644/serverless-the-future-of-cloud-computing.html.

[26] Amazon web services, «Serverless,» 2020. [En línea]. Available: https://aws.amazon.com/es/serverless/.

[27] N. Dabit, Full Stack Serverless, Sebastopol: O'REILLY, 2020.

[28] C. Gurturk, Building Serverless Architectures, Birmingham: O'REILLY, 2017.

[29] P. Castro, V. Ishakian, V. Muthusamy y A. Slominski, «The Rise of Serverless Computing,» *Communications of the ACM,* vol. 62, nº 12, pp. 44-54, December 2019.

[30] P. Goldstein, «What Is Serverless Computing, and How Can It Benefit Your Team?,» 3 June 2019. [En línea]. Available: https://fedtechmagazine.com/article/2019/06/what-serverless-computing-and-how-can-it-benefit-your-team-perfcon.

[31] M. E. T. H. a. G. W. J. Manner, «Cold Start Influencing Factors in Function as a Service,» *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion),* pp. 181-188, 2018.

[32] V. I. V. M. a. A. S. P. Castro, «Serverless Programming (Function as a Service),» *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS),* pp. 2658-2659, 2017.

[33] G. C. Fox, V. Ishakian, V. Muthusamy y A. Slominski, «Status of Serverless Computing and Function-as-a-Service(FaaS) in industry and Research,» 2017.

[34] A. I. S. S. a. M. T. Erwin van Eyk, «The SPEC cloud group's research vision on FaaS and serverless architectures,» *In Proceedings of the 2nd International Workshop on Serverless Computing (WoSC '17),* pp. 1-4, 2017.

[35] B. I. e. al., «Serverless Computing: Current Trends and Open Problems,» *Chaudhary S., Somani G., Buyya R. (eds) Research Advances in Cloud Computing.,* 2017.

[36] P. C. S. J. F. N. M. V. M. R. R. P. S. a. O. T. Ioana Baldini, «The serverless trilemma: function composition for serverless computing.,» *In Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017),* pp. 89-103, 2017.

[37] Microsoft Azure, «Azure Functions,» 2020. [En línea]. Available: https://azure.microsoft.com/es-es/services/functions/#features.

[38] Google Cloud, «Cloud Functions,» 2020. [En línea]. Available: https://cloud.google.com/functions/?utm_source=google&utm_medium=cpc&utm_campaign=emea-es-all-en-dr-bkws-all-all-trial-e-gcp-1008073&utm_content=text-ad-none-any-DEV_c-CRE_253523426609-ADGP_Hybrid+%7C+AW+SEM+%7C+BKWS+~+EXA_M:1_ES_EN_General_Cloud+Function.

[39] C. Hernando, «¿Qué es serverless?,» 2017 April 17. [En línea]. Available: https://chernando.xyz/articulos/que-es-serverless-baas-y-faas/.

[40] FayerWayer, «El origen de: El Cómputo en la Nube,» 7 January 2012. [En línea]. Available: https://www.fayerwayer.com/2012/01/el-origen-de-el-computo-en-la-nube/.

# Glosario

## *A*

*Amazon Web Services*: Amazon Web Services es una colección de servicios de computación en la nube pública que en conjunto forman una plataforma de computación en la nube. · 8

*API*: 'Application Programming Interface' es un conjunto de reglas (código) y especificaciones que las aplicaciones pueden seguir para comunicarse entre ellas · 30

*API Gateway*: (https://aws.amazon.com/es/api-gateway/) es un servicio completamente administrado que facilita a los desarrolladores la creación, la publicación, el mantenimiento, el monitoreo y la protección de API a cualquier escala. · 21

*Auth0*: (https://auth0.com/) aplicación que provee un servicio de autenticación para webs, mobile y aplicaciones. · 25

AWS *ECS*: (https://aws.amazon.com/es/ecs/) Amazon Elastic Container Service, es un servicio de orquestación de contenedores completamente administrado. · 22

AWS *EKS*: (https://aws.amazon.com/es/eks/) Amazon Elastic Kubernetes Service (Amazon EKS) es un servicio Kubernetes completamente administrado. Clientes como Intel, Snap, Intuit, GoDaddy y Autodesk usan EKS para ejecutar sus aplicaciones más sensibles y de misión crítica debido a su seguridad, confiabilidad y escalabilidad. · 22

*AWS Kinesis*: (https://aws.amazon.com/es/kinesis/) facilita la recopilación, el procesamiento y el análisis de datos de streaming en tiempo real para obtener datos de manera oportuna y reaccionar rápidamente ante información nueva. · 18

*AWS Lambda*: (https://aws.amazon.com/es/lambda/) AWS Lambda es el principal servicio function-as-a-service de AWS. Es una plataforma informática sin servidor basada en eventos proporcionada por Amazon Web Services. · 8

*AWS x-ray*: (https://aws.amazon.com/es/xray/) servicio proporcionado por *AWS* que revisa y prevee errores. · 34

*Azure WebJobs*: es un servicio dentro de Azure que ejecuta jobs, que a su vez son un conjunto coherente de instrucciones para realizar un trabajo particular. · 12

## *B*

*Back end*: el back end es la parte del desarrollo que se encarga de que toda la lógica de la aplicación y se comunica con el front end. · 9

*Back end as a service (BaaS)*: es un modelo que permite a los desarrolladores web y desarrolladores mobile una serie de servicios permitiendo prescindir totalmente de una api personalizada. · 8

*Big data*: conjuntos de datos o combinaciones de conjuntos de datos cuyo tamaño (volumen), complejidad (variabilidad) y velocidad de crecimiento (velocidad) dificultan su captura, gestión, procesamiento o análisis mediante tecnologías y herramientas convencionales. · 19

## *C*

*C*: es un lenguaje de programación de propósito general2; 1 originalmente desarrollado por Dennis Ritchie entre 1969 y 1972 en los Laboratorios Bell,1 como evolución del anterior lenguaje B, a su vez basado en BCPL. · 14

Centros de procesamiento de datos: Se denomina centro de procesamiento de datos (CPD) al espacio donde se concentran los recursos necesarios para el procesamiento de la información de una organización. · 1

*Cloudwatch logs*: (https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html) servicio de alamacenamiento de logs de *AWS.* · 34

## *E*

*Endpoint*: un punto final de comunicación es un tipo de nodo de red de comunicación. · 21

## *F*

*framework*: un entorno de trabajo, o marco de trabajo es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar. · 8

*Front end*: el front end es la parte del software que interactúa con los usuarios · 9

## *G*

*GitHub*: (https://github.com/) GitHub es una forja para alojar proyectos utilizando el sistema de control de versiones Git. · 12

## *H*

*Hardware*: conjunto de elementos físicos o materiales que constituyen una computadora o un sistema informático. · 24

*HTML*: HTML, siglas en inglés de HyperText Markup Language, hace referencia al lenguaje de marcado para la elaboración de páginas web. · 14

HTTP: el Protocolo de transferencia de hipertexto (en inglés, Hypertext Transfer Protocol, abreviado HTTP) es el protocolo de comunicación que permite las transferencias de información en la World Wide Web. · 18

## *I*

*IEEE*: (Institute of Electrical and Electronics Engineers,en castellano, Instituto de Ingenieros Eléctricos y Electrónicos) es un organismo que entre otras cosas se encarga de hacer estándares de protocolos como los de conexión entre equipos. · 9

*Infrastructure as a service (IaaS)*: IaaS (Infrastructure as a Service) es un servicio informático en la nube que da acceso a una infraestructura de TI altamente flexible a través de Internet. · 7

## *J*

*java*: (https://www.java.com/es/) es un lenguaje de programación y una plataforma informática comercializada por primera vez en 1995 por Sun Microsystems. · 14

*Javascript*: (https://www.javascript.com/) es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico. · 14

*Jenkins*: (https://www.jenkins.io/) ayuda en la automatización de parte del proceso de desarrollo de software mediante integración continua y facilita ciertos aspectos de la entrega continua. · 24

*JSON*: (JavaScript Object Notation) es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript. · 21

## *K*

*Keep Alives*: es un mensaje enviado por un dispositivo a otro para verificar que el enlace entre los dos está funcionando, o para evitar que el enlace se rompa. En el caso de serverless se usa para que el proveedor no retire el contenedor con la aplicación. · 20

*Kubernetes*: (https://kubernetes.io) Kubernetes es un sistema de código libre para la automatización del despliegue, ajuste de escala y manejo de aplicaciones en contenedores que fue originalmente diseñado por Google y donado a la Cloud Native Computing Foundation. · 12

## *M*

*Mesos*: (http://mesos.apache.org/documentation/latest/containerizers/) es un kernel administrador de clúster de código abierto desarrollado en la Universidad de California, Berkeley. Mesos corre en cada nodo del cluster y provee aplicaciones (como Hadoop, Spark, Kafka entre otras) con API's para el manejo de recursos y planificación de tareas de todo el datacenter. · 22

## *N*

*Node.js*: (https://nodejs.org/es/) es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. · 12

## *O*

*on-premise*: o en local se refiere al tipo de instalación de una solución de software. Esta instalación se lleva a cabo dentro del servidor y la infraestructura (TIC) de la empresa. · 12

*Open source*: el software de código abierto es el software cuyo código fuente y otros derechos que normalmente son exclusivos para quienes poseen los derechos de autor, son publicados bajo una licencia de código abierto o forman parte del dominio público. · 12

*OpenStack Picasso*: API para servicios funtion-as-a-service, abstrae la capa de infraestructura y ofrece simplicidad, eficiencia y escalabilidad para desarrolladores y operaciones. · 12

## *P*

*Platform as a service (PaaS)*: Es una categoría de los servicios de computación cloud que provee una plataforma que permite a los clientes desarrollar, correr y administrar aplicaciones sin la complejidad de construir y mantener una infraestructura típica asociada a desarrollar y desplegar una aplicación. · 8

*Postgresql*: (https://www.postgresql.org/) también llamado Postgres, es un sistema de gestión de bases de datos relacional orientado a objetos y de código abierto, publicado bajo la licencia PostgreSQL. · 14

proveedor de servicio: un proveedor de servicios es una entidad que presta servicios a otras entidades. Por lo general, esto se refiere a un negocio que

ofrece la suscripción o servicio web a otras empresas o particulares. · 8
*Python*: (https://www.python.org/) es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. · 14

## *R*

*Rundeck*: (https://www.rundeck.com/open-source) es una herramienta de automatización, de código abierto para uso en centros de datos o en la nube. · 24

## *S*

SAR: (Serverless Application Repository) (https://aws.amazon.com/es/serverless/serverlessrepo/) es un servicio de control de versiones de Funciones Lambda · 23
Servicios *cloud*: la computación en la nube es la disponibilidad a pedido de los recursos del sistema informático, especialmente el almacenamiento de datos y la capacidad de cómputo. · 8
Servidores *Rack*: son equipos de gama alta, estandarizados, que se incluyen dentro de bastidores, permitiendo una mejor organización y hacer cambios sin reiniciar todo el sistema. · 1
*Software as a service (SaaS)*: es un modelo de distribución de software donde el soporte lógico y los datos que maneja se alojan en servidores de una compañía de tecnologías de información y comunicación, a los que se accede vía Internet desde un cliente. · 8
*Startup*: los términos empresa emergente, startup, compañía emergente, compañía de arranque y compañía incipiente se utilizan en el mundo empresarial aplicados a empresas de reciente creación, normalmente fundadas por un emprendedor o varios, sobre una base tecnológica, innovadoras y supuestamente con una elevada capacidad de rápido crecimiento · 3

## *T*

Tecnologías de tiempo compartido: en computación, el uso del tiempo compartido (calco semántico del inglés time-sharing) se refiere a compartir de forma concurrente un recurso computacional (tiempo de ejecución en la CPU, uso de la memoria, etc.) entre muchos usuarios · 8
*tenant*: conjunto de computación que se asigna a un cliente dentro de un clustes de instancias. · 12

## *V*

*VPCs*: permite aprovisionar una sección de la nube de AWS aislada de forma lógica, en la que puede lanzar recursos de AWS en una red virtual que usted defina. · 20

## *W*

*Workaround*: tipo de resolución de un problema que no suele ser la forma adecuada de resolverlo, similar a un apaño. · 34
WWW: hace referencia a la World Wide Web (WWW) o red informática mundial1 es un sistema de distribución de documentos de hipertexto o hipermedia interconectados y accesibles a través de Internet. · 14

# Anexos

## A. Pruebas de concepto y resultados

En este apartado se muestran algunas de las pruebas de concepto realizadas durante el trabajo para probar la veracidad del comportamiento de las funciones *lambda* y las características que se indican en la documentación por el proveedor de servicio y algunas fuentes menos fiables. Algunas de estas pruebas no se han documentado, debido a que se tratan de pruebas de concepto que ya se tienen en cuenta a lo largo del trabajo, aún así se han documentado las pruebas básicas relacionadas con las principales características de las arquitecturas *serverless.*
Cabe destacar, que hay algunas pruebas que debido a su alta complejidad salen del alcance del trabajo, como son las pruebas relacionadas con costes, por ello en los apartados que se tratan los costes se hace referencia a las investigaciones usadas. También hay algunas pruebas que necesitan un presupuesto para no salirse del *Free Tier*, esto es el servicio mínimo ofrecido por *AWS* sin coste adicional, como las pruebas relacionadas con el apartado de denegación de servicio 6.7.2.1.

### Arranques en frío y en caliente

En primer lugar, se va a realizar una prueba de concepto sobre una de las principales características de las *arquitecturas serverless,* que son los arranques en frío y en caliente. Los arranques en frío se producen al no haberse usado en un periodo de tiempo largo la función, lo que provoca que el proveedor retire el contenedor donde se ejecuta esa función, por lo que al volver a llamarla se produce un retardo mayor en la ejecución al tener que volver a crear el contenedor y el entorno donde va a correr esta, además variará en función de la cantidad de librerías que se importen, variables de entorno que se usen, *VPC*s, sistemas de ficheros y conexiones a bases de datos. Por el contrario, los arranques en caliente se producen cuando entre dos ejecuciones de la misma función no ha pasado el tiempo suficiente para que el proveedor de servicio no retire el contenedor, estos son más rápidos que los arranques en frío y por lo tanto el escenario ideal.

Para ello se ha preparado una función *Lambda* muy simple, al tratarse de una prueba de concepto que quiere comprobar los tiempos de retardo se producen y compararlos con los proporcionados por *AWS* en este caso. No hace falta ningún programa específico, por lo que se ha utilizado la plantilla que proporciona *AWS* de un programa 'hola mundo' usando *Python 3.8*, tal como se presenta en la siguiente imagen.

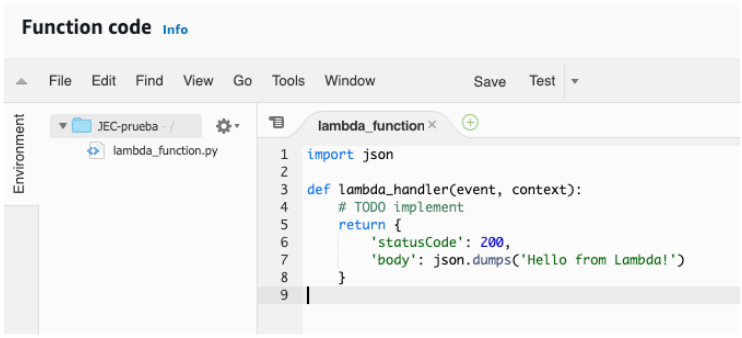



Figura 9. Código de la función a probar latencia de arranque

*AWS* en la documentación [26] correspondiente a arranques en frío no especifica los tiempos que producen los arranques en frío sólo indica que se latencia surge cuando se invoca una función por primera vez, cuando se actualiza o después de cierto tiempo sin utilizarla, por lo que vamos a probar que es así.

A continuación, se ha ejecutado la función diez veces seguidas para comprobar si, como es de esperar, la primera ejecución ha resultado ser un arranque el frío y el resto se tratan de arranques en caliente. En la tabla 4 se puede comprobar los registros que ha dejado la función, con la duración de cada ejecución, de igual forma se puede comprobar en la tabla 5 el tiempo de ejecución de cada función.

| Timestamp | Message |
|---|---|
| 2020-07-07T14:14:06.499+02:00 | REPORT RequestId: 683207ec-9c89-43de-8ec0-e0ece47b1bc5 Duration: 1.49 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB Init Duration: 131.12 ms |
| 2020-07-07T14:14:25.316+02:00 | REPORT RequestId: c0e1c86b-4167-4569-bb5f-4af3e590de54 Duration: 0.91 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB |
| 2020-07-07T14:14:31.308+02:00 | REPORT RequestId: c9f9d0ae-eaad-402c-b54b-fca53c8b1ff1 Duration: 1.09 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB |
| 2020-07-07T14:14:43.365+02:00 | REPORT RequestId: d7c1e300-bc9e-40a7-8ed6-61b7833c607b Duration: 1.31 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB |
| 2020-07-07T14:14:53.407+02:00 | REPORT RequestId: 8f247384-b28a-44c6-aee8-9f5694cae638 Duration: 1.30 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB |
| 2020-07-07T14:15:03.154+02:00 | REPORT RequestId: bc5de867-2595-4f4b-b067-89d5315adeaf Duration: 1.03 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB |
| 2020-07-07T14:15:10.629+02:00 | REPORT RequestId: a86014f4-04c3-4e10-a96a-f677c847ff43 Duration: 0.99 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB |
| 2020-07-07T14:15:56.088+02:00 | REPORT RequestId: cc0c0ab3-a20a-43c5-b2a7-1a25f554623f Duration: 1.07 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 50 MB |
| 2020-07-07T14:16:23.296+02:00 | REPORT RequestId: 1b6e88f8-3feb-43c7-b0f4-d75bcacb2254 Duration: 1.08 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 51 MB |
| 2020-07-07T14:16:34.601+02:00 | REPORT RequestId: 1ab04b59-3e62-4f09-a855-6cad143a3289 Duration: 1.07 ms Billed Duration: 100 ms Memory Size: 128 MB Max Memory Used: 51 MB |

Tabla 4. Registros de las 10 ejecuciones de prueba.

Recent invocations

| # | Timestamp | RequestID | LogStream | DurationInMS | BilledDurationInMS | MemorySetInMB | MemoryUsedInMB |
|---|---|---|---|---|---|---|---|
| 1 | 2020-07-07T12:16:23.296Z | 1b6e88f8-3feb-43c7-b0f4-d75bcacb2254 | 2020/07/07/[$LATEST]bc794c44c8594c38b153d828021e09f2 | 1.08 | 100 | 128 | 51 |
| 2 | 2020-07-07T12:15:56.088Z | cc0c0ab3-a20a-43c5-b2a7-1a25f554623f | 2020/07/07/[$LATEST]bc794c44c8594c38b153d828021e09f2 | 1.07 | 100 | 128 | 50 |
| 3 | 2020-07-07T12:15:10.629Z | a86014f4-04c3-4e10-a96a-f677c847ff43 | 2020/07/07/[$LATEST]bc794c44c8594c38b153d828021e09f2 | 0.99 | 100 | 128 | 50 |
| 4 | 2020-07-07T12:15:03.154Z | bc5de867-2595-4f4b-b067-89d5315adeaf | 2020/07/07/[$LATEST]bc794c44c8594c38b153d828021e09f2 | 1.03 | 100 | 128 | 50 |
| 5 | 2020-07-07T12:14:53.407Z | 8f247384-b28a-44c6-aee8-9f5694cae638 | 2020/07/07/[$LATEST]bc794c44c8594c38b153d828021e09f2 | 1.3 | 100 | 128 | 50 |
| 6 | 2020-07-07T12:14:43.365Z | d7c1e300-bc9e-40a7-8ed6-61b7833c607b | 2020/07/07/[$LATEST]bc794c44c8594c38b153d828021e09f2 | 1.31 | 100 | 128 | 50 |
| 7 | 2020-07-07T12:14:31.308Z | c9f9d0ae-eaad-402c-b54b-fca53c8b1ff1 | 2020/07/07/[$LATEST]bc794c44c8594c38b153d828021e09f2 | 1.09 | 100 | 128 | 50 |
| 8 | 2020-07-07T12:14:25.316Z | c0e1c86b-4167-4569-bb5f-4af3e590de54 | 2020/07/07/[$LATEST]bc794c44c8594c38b153d828021e09f2 | 0.91 | 100 | 128 | 50 |
| 9 | 2020-07-07T12:14:06.499Z | 683207ec-9c89-43de-8ec0-e0ece47b1bc5 | 2020/07/07/[$LATEST]bc794c44c8594c38b153d828021e09f2 | 1.49 | 100 | 128 | 50 |

Tabla 5.Tabla resumen de las características de las pruebas.

En la tabla 4, que corresponde los registros que han dejado las ejecuciones de las funciones podemos observar que en la primera ejecución se produce un arranque en frío, el cual dura 131,12ms y el tiempo de la ejecución de la función es de 1,49ms, lo que suma un total de 132,61ms hasta que la función termina. El resto no tienen latencia de arranque y su duración es de media 1,09444ms, lo que está muy por debajo del tiempo total de la primera ejecución y por debajo del tiempo de ejecución de la primera función.

También se puede comprobar en la siguiente gráfica que representa el tiempo mínimo, máximo y medio de ejecución de la función, y se puede observar que la primera ejecución dura más que el resto, debido al arranque en frío, aunque en esta gráfica no se representa todo el tiempo invertido en este arranque, ya que solo tiene en cuenta la duración de la ejecución, si no el pico sería mucho más elevado.

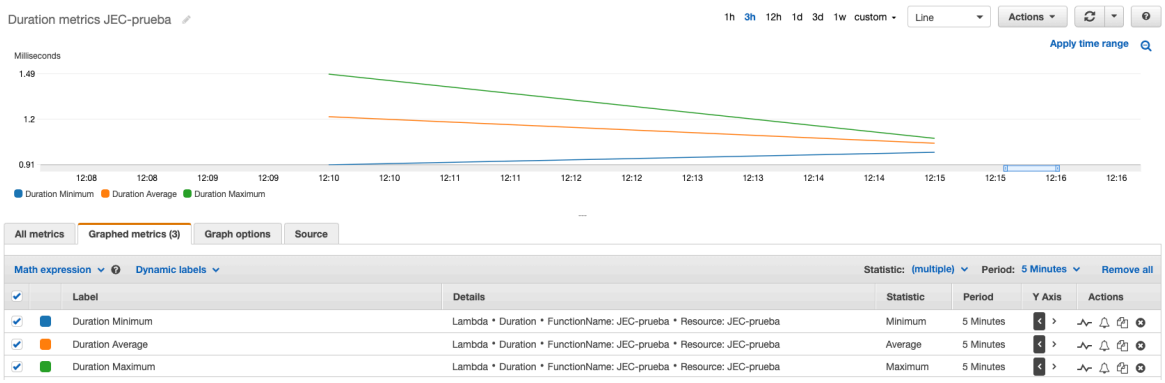



Figura 10. Gráfica de duración máxima, media y mínima de la ejecución.

El proveedor al no dar muchos detalles sobre los tiempos que manejan los arranques en frío podemos concluir que funcionan tal y como se describe, después de crear la función se produce un arranque en frío, de igual manera cuando se vuelve a probar después de 45 minutos.

## Duración de la ejecución

La duración de la ejecución es una característica importante y a tener en cuenta cuando se trabaja con arquitecturas *serverless* ya que el sistema, al estar administrado por el proveedor limita la duración de la ejecución de un programa, limitando los casos de uso de estas arquitecturas, aunque hay formas de orquestar funciones esta solución no suele ser recomendable.

*AWS* respecto a la duración de las ejecuciones de funciones *lambda* [26], tal y como se ha descrito a lo largo del trabajo, explica que estas se pueden limitar al tiempo que se desee con un máximo de 15 minutos de ejecución. A continuación, se va a probar si estos datos especificados por el proveedor son ciertos. Para ello se ha usado un código simple con un bucle infinito y un contador que muestra los minutos que van pasando, se probará con un límite configurado y con el máximo permitido a ver si se cumplen las especificaciones.

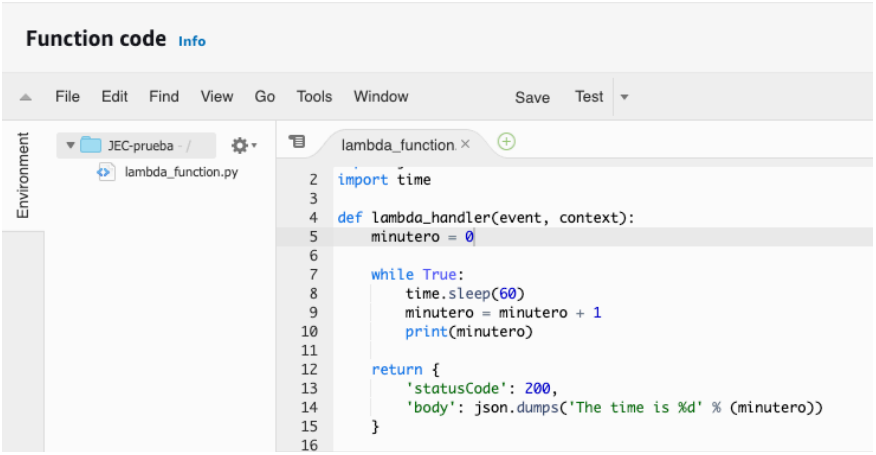



Figura 11. Código de prueba para comprobar la duración de la ejecución de una función *lambda*.

A continuación, se ha establecido un límite de 3 segundos para probar que se puede limitar la función al tiempo máximo de ejecución deseado. Tal y como ha indicado el proveedor la función se detuvo por un *timeout* a los 3003,22 ms lo que implica que la función ha durado 3,22 ms más de lo que debería, tal como indica el registro.


Tabla 6. Registro de la ejecución de la prueba con un límite de 3 segundos.

Se va a probar a forzar el límite al máximo que permite el proveedor, 15 minutos. La función se ha parado, de nuevo, por un *timeout* a los 900085,56 ms, lo que significa que ha durado 85,56 ms más de lo que se indica que va a durar, tal como infica la Tabla 7.

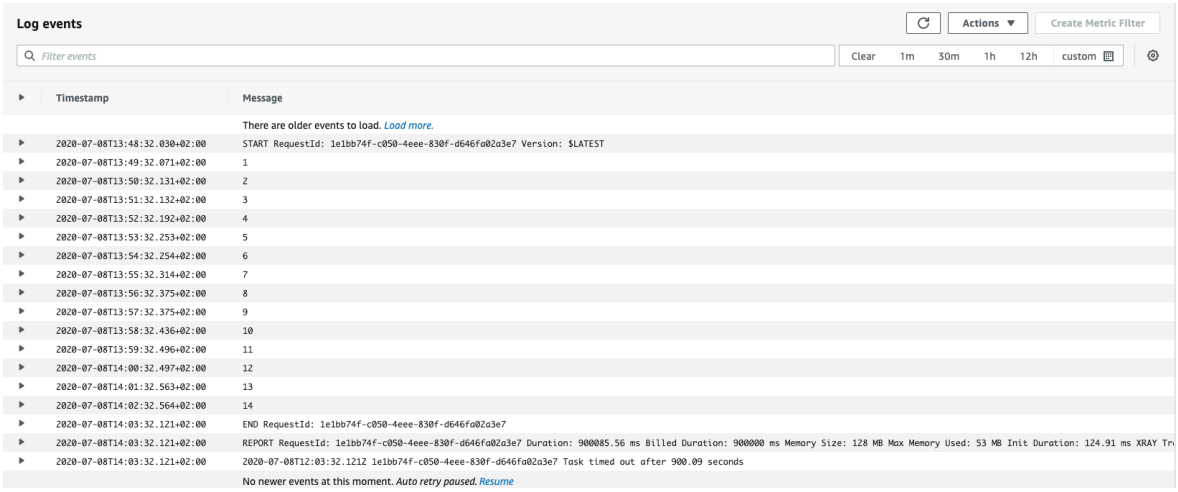



Tabla 7. Registro de la ejecución de la prueba con un límite de 15 minutos.

Recent invocations

| # | Timestamp | RequestID | LogStream | DurationInMS | BilledDurationInMS | MemorySetInMB | MemoryUsedInMB |
|---|---|---|---|---|---|---|---|
| 1 | 2020-07-08T12:03:32.121Z | 1e1bb74f-c050-4eee-830f-d646fa02a3e7 | 2020/07/08/[$LATEST]8229636bfbc74c37b4b4be5e30170c3a | 900085.56 | 900000 | 128 | 53 |
| 2 | 2020-07-08T11:46:09.919Z | 8d6054ba-3276-46ac-89d0-5f78a30560e8 | 2020/07/08/[$LATEST]5d2c9457cd964235b240fdd8810f16a7 | 60063.16 | 60100 | 128 | 53 |
| 3 | 2020-07-08T11:44:26.375Z | 790ee969-79a5-4686-bc0b-e6d18b071215 | 2020/07/08/[$LATEST]2436efa529c94fb186917cfb63fbef8e | 60063.51 | 60100 | 128 | 53 |
| 4 | 2020-07-08T11:41:13.864Z | 89f836ea-b5da-493b-b253-0c044d138065 | 2020/07/08/[$LATEST]c4363246e6084b14b625d6ed35184d65 | 3003.22 | 3000 | 128 | 53 |
| 5 | 2020-07-08T11:39:45.145Z | 07452957-9abf-4ef5-b512-2f2351cccf0e | 2020/07/08/[$LATEST]f178b2f1a8fb4f4d93411c195aa7ab1b | 16.41 | 100 | 128 | 50 |
| 6 | 2020-07-08T11:38:29.382Z | c5386cc6-ba2c-4ffd-b2e6-35ac4960153b | 2020/07/08/[$LATEST]7ef517e4506447d78817891dda1ea1da | 10.07 | 100 | 128 | 50 |

Tabla 8. Tabla resumen de la ejecución de la prueba con un límite de 15 minutos.

Se puede concluir, que el tiempo de duración de una ejecución lambda, tal y como indica *AWS,* aunque no es exacto y se excede unos milisegundos de más.

## Estado

Las arquitecturas serverless, en concreto los servicios *FaaS* son servicios que no conservan el estado, esto significa que de una sesión a otra las variables, conexiones y demás no tienen por qué valer lo mismo y no se debe contar con que estas mantienen el valor. Así lo dice en su documentación *AWS* [26] y la única forma de guardar algún dato es en la ruta /tmp, la cual tiene una memoria muy limitada.

Para probar esto se ha realizado un simple código con dos variables que van cambiando su valor y se ha llamado a esta función dos veces seguidas. En la figura 12 se muestra el código usado.

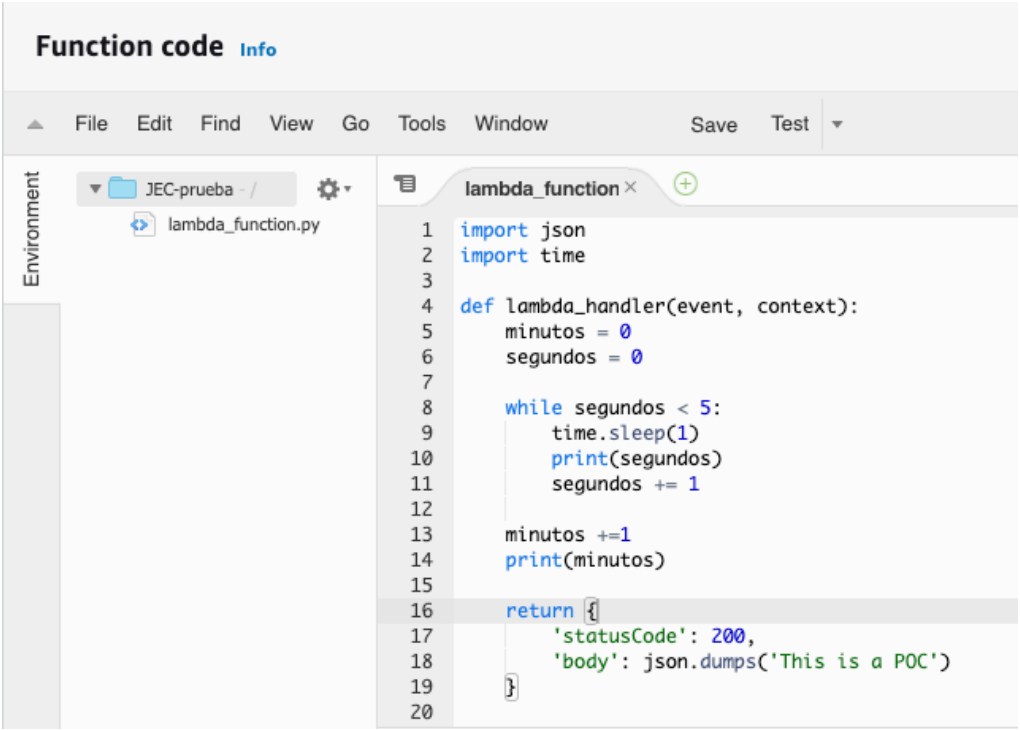



Figura 12. Código usado para probar si se tratan de funciones sin estado.

En las siguientes imágenes se muestran los dos registros de ejecución de la función que se han ejecutado de forma consecutiva. Se puede comprobar en el registro que las variables tienen los mismos valores en las dos invocaciones de la misma función por lo que

las variables no han conservado el mismo valor a pesar de haberse producido dos invocaciones consecutivas.

**Log events**

Filter events

| Timestamp | Message |
|---|---|
| | There are older events to load. Load more. |
| 2020-07-08T21:07:14.538+02:00 | START RequestId: 7dd73627-da1f-427a-83fa-6ec3e14e671d Version: $LATEST |
| 2020-07-08T21:07:15.557+02:00 | 0 |
| 2020-07-08T21:07:16.558+02:00 | 1 |
| 2020-07-08T21:07:17.559+02:00 | 2 |
| 2020-07-08T21:07:18.560+02:00 | 3 |
| 2020-07-08T21:07:19.562+02:00 | 4 |
| 2020-07-08T21:07:19.562+02:00 | 1 |
| 2020-07-08T21:07:19.563+02:00 | END RequestId: 7dd73627-da1f-427a-83fa-6ec3e14e671d |
| 2020-07-08T21:07:19.563+02:00 | REPORT RequestId: 7dd73627-da1f-427a-83fa-6ec3e14e671d Duration: 5024.16 ms |

Tabla 9. Registro de la primera ejecución probando el estado de *AWS Lambda*.

| | |
|---|---|
| 2020-07-08T21:09:40.869+02:00 | START RequestId: f3a8b11a-d464-474f-8a55-2125d1b3fa80 Version: $LATEST |
| 2020-07-08T21:09:41.873+02:00 | 0 |
| 2020-07-08T21:09:42.874+02:00 | 1 |
| 2020-07-08T21:09:43.876+02:00 | 2 |
| 2020-07-08T21:09:44.877+02:00 | 3 |
| 2020-07-08T21:09:45.878+02:00 | 4 |
| 2020-07-08T21:09:45.878+02:00 | 1 |
| 2020-07-08T21:09:45.879+02:00 | END RequestId: f3a8b11a-d464-474f-8a55-2125d1b3fa80 |
| 2020-07-08T21:09:45.879+02:00 | REPORT RequestId: f3a8b11a-d464-474f-8a55-2125d1b3fa80 Duration: 5007.02 ms |

Tabla 10.Registro de la segunda ejecución probando el estado de AWS Lambda.

Se puede concluir que *AWS* no miente en su documentación cuando se explica que las funciones *lambda* son funciones sin estado y por ello no se debe tener en cuenta a la hora de trabajar con ellas que lo mantienen.

## Escalado automático

El escalado en las arquitecturas *serverless* gracias a los servicios *FaaS* es automático, y así lo indica AWS [26]. Por lo que para probar si el escalado es automático se ha creado una función simple que sirva como de prueba y se va a ejecutar de forma concurrente varias veces la función. En la siguiente figura se muestra el código empleado para esta prueba.

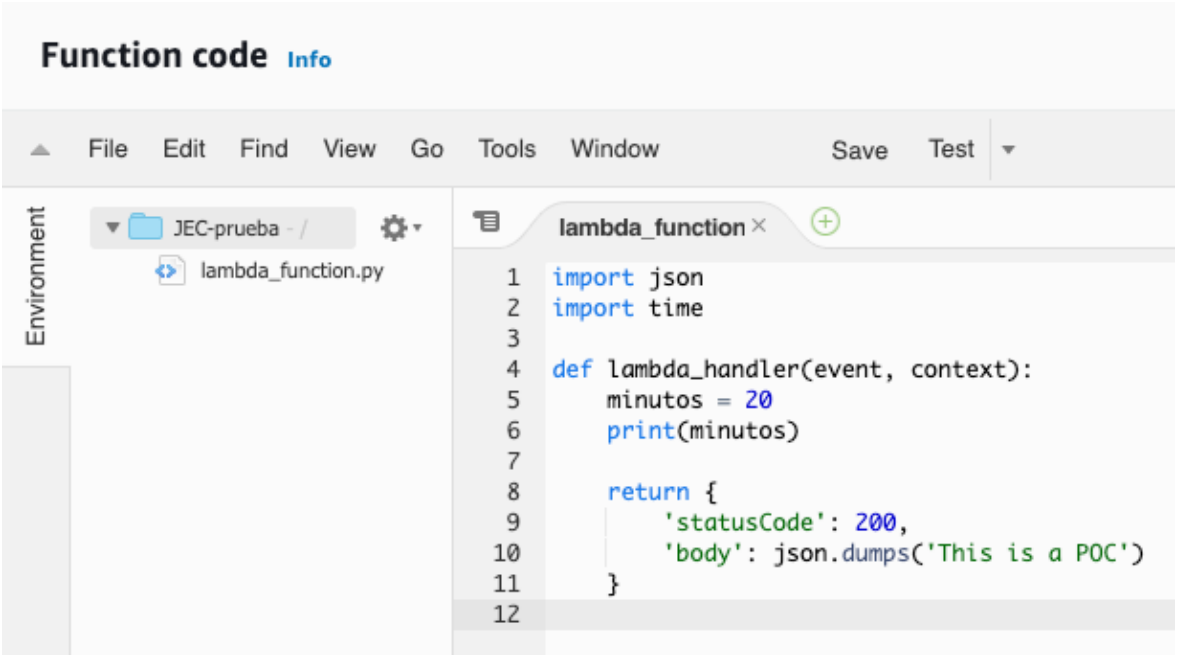
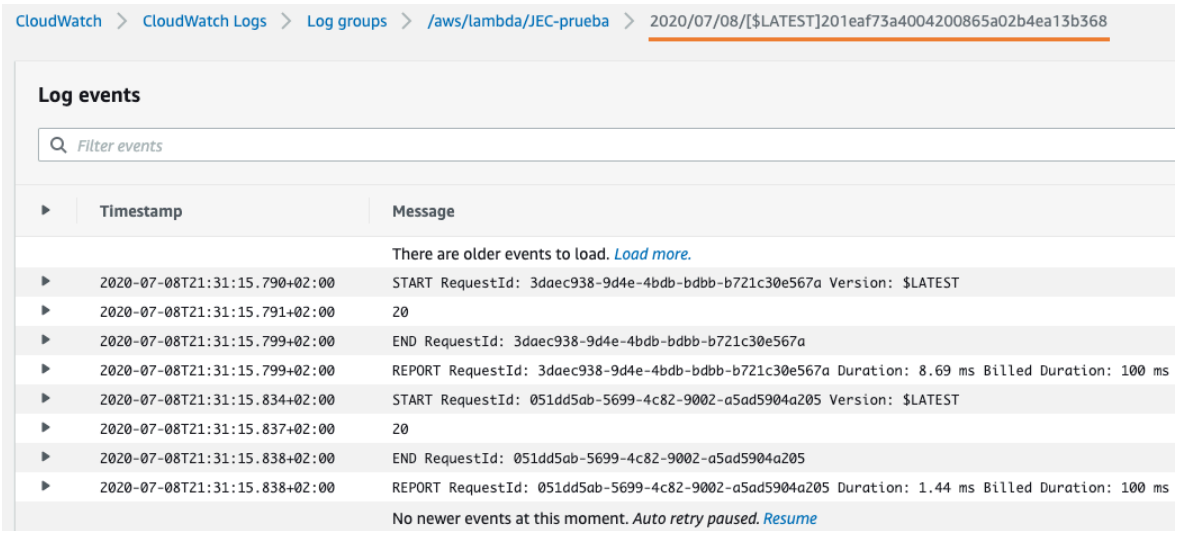



Figura 13. Código usado para la prueba de escalado.

A continuación, se muestran los registros de las dos ejecuciones que se han ejecutado en diferentes contendores como se puede ver en la parte superior que indica la ruta que se remarca en rojo, además de que la hora de los registros es casi similar.




| Timestamp | Message |
|---|---|
| | There are older events to load. *Load more.* |
| 2020-07-08T21:31:15.790+02:00 | START RequestId: 3daec938-9d4e-4bdb-bdbb-b721c30e567a Version: $LATEST |
| 2020-07-08T21:31:15.791+02:00 | 20 |
| 2020-07-08T21:31:15.799+02:00 | END RequestId: 3daec938-9d4e-4bdb-bdbb-b721c30e567a |
| 2020-07-08T21:31:15.799+02:00 | REPORT RequestId: 3daec938-9d4e-4bdb-bdbb-b721c30e567a Duration: 8.69 ms Billed Duration: 100 ms |
| 2020-07-08T21:31:15.834+02:00 | START RequestId: 051dd5ab-5699-4c82-9002-a5ad5904a205 Version: $LATEST |
| 2020-07-08T21:31:15.837+02:00 | 20 |
| 2020-07-08T21:31:15.838+02:00 | END RequestId: 051dd5ab-5699-4c82-9002-a5ad5904a205 |
| 2020-07-08T21:31:15.838+02:00 | REPORT RequestId: 051dd5ab-5699-4c82-9002-a5ad5904a205 Duration: 1.44 ms Billed Duration: 100 ms |
| | No newer events at this moment. *Auto retry paused.* *Resume* |

Tabla 11. Registro de la primera ejecución concurrente de la prueba de escalado horizontal.

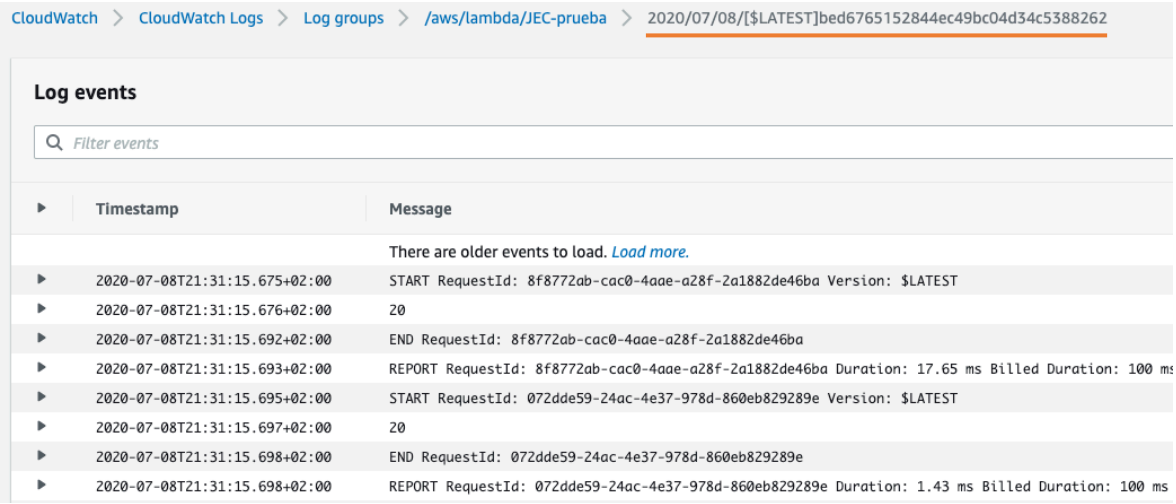



Tabla 12. Registro de la segunda ejecución concurrente de la prueba de escalado horizontal.

Por último, se ha comprobado la métrica de *concurrent executions* que muestra el número de ejecuciones concurrentes de la función *Lambda.* Aunque se ve de forma discreta el punto en la parte superior derecha indica que ha habido 3 ejecuciones concurrentes.

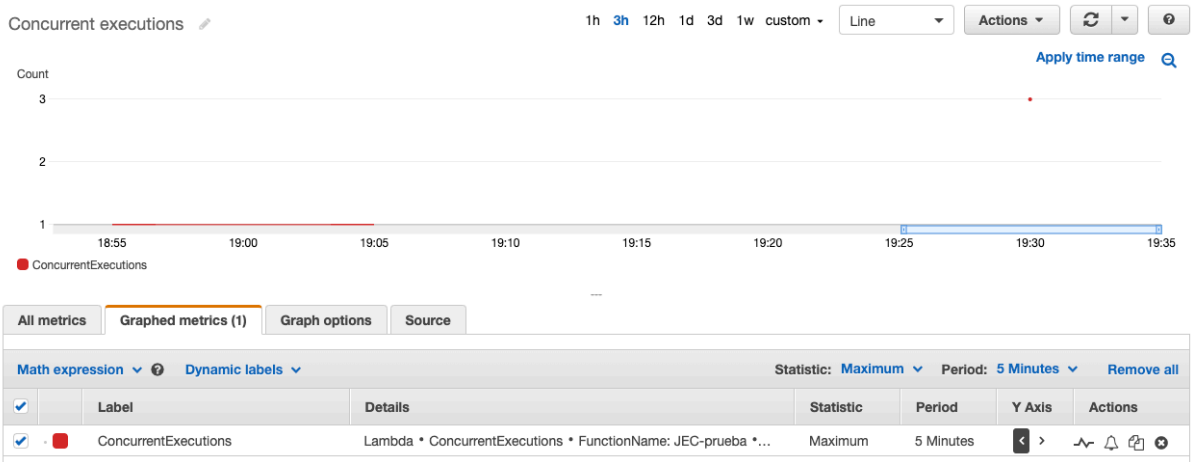



Figura 13. Gráfica de ejecuciones concurrentes.

Se puede concluir con estas pruebas que tal como indica *AWS* el escalado es automático, aunque con un matiz, por defecto no escala horizontalmente, es necesario configurar el parámetro *reserve concurrency* con el número máximo de instancias que se pueden paralelizar y por defecto este está a 0 con un máximo de 1000.